COMP 426 Midterm 1 Study Guide

Modern Web Programming · UNC Fall 2026 · Covers L01–L06 (incl. L05.5) and readings R00–R03

0. Exam logistics and how to use this guide

When. Monday, September 14, 2026, in lecture (3:35–4:50pm, Sitterson 014). You get the full 75 minutes.

Format. On paper and closed-book. Worth 10% of the course grade (the two midterms together are 20%). Collaboration is prohibited.

Scope. Everything through L06 (Asynchronous TypeScript). The instructor said in L06: "Today's lecture is the last content to be covered on the exam." Question format was not announced as of L06.

What is in scope

Lecture	Date	Core ideas you must be able to explain
L01 The Internet	Aug 17	WWW vs Internet vs network; IP addresses; protocols and layering; TCP/IP; client/server; HTTP request/response; HTTPS; URL anatomy; DNS; the full "type a URL, get a page" sequence.
L02 HTML and CSS	Aug 19	Elements, tags, nesting, attributes; head vs body; class vs id; CSS rules, selectors (simple, compound, descendant, child, sibling, pseudo-class); reading compound selectors right to left; box model and total width; layout viewport; display modes; flexbox basics; "HTML or CSS determines layout?" (both).
L03 JS, TS, Build Systems	Aug 24	How JS loads with a page; dynamic vs static typing; TypeScript as a superset; transpilation; build tools (Vite); Node.js as a runtime; npm; dev server, localhost, ports; connection to gcc/Maven from earlier courses.
L04 Events and the DOM	Aug 26	The DOM tree (Document, documentElement, elements, text nodes, attributes); drawing DOM trees; `document` API for finding and navigating nodes; `addEventListener`; innerHTML vs innerText; the event object and its universal properties.

L05 Functional Programming	Aug 31	Memory diagrams in the instructor's notation (stack frames F#, P, RA, RV; heap function objects with ids); tracing calls; higher-order functions; closures; functional vs object-oriented paradigms; objects from closures; getters/setters.
L05.5 HTTP and the Rise of Apps	Sep 2	HTML vs HTTP; TCP connection and ports 80/443; statelessness; request line, headers, body; methods; status line and code classes; key headers (Host, User-Agent, Accept, Cookie); forms and CGI; cookies; AJAX/XHR; web as an application framework; fetch and promises as the modern approach.
L06 Async TypeScript	Sep 9	API endpoints and JSON; single-threaded JS and blocking; callbacks; promises and `.then`/`.catch` chaining; `async`/`await`; predicting console output order; `fetch` and `.json()`.
Readings R00–R03	—	Syllabus policies; HTML/CSS fundamentals; TypeScript syntax for Java developers (types, arrays, loops, functions, classes, interfaces, structural typing, enums, type aliases, unions, generics, spread/rest, destructuring); higher-order functions (functions as values, function types, passing and returning functions).

How to study with this guide

- **Each lecture section** follows the slides in order, expands the terse bullets into full explanations, and ends with "Check yourself" questions. Answers are hidden under a toggle so you can self-test.
- **Boxes:** blue "Key idea" boxes are the definitions most likely to be asked verbatim. Orange "Watch out" boxes flag common mistakes. Green "Exam tip" boxes tell you how a question on that topic is likely to be phrased.
- **Practice exam** at the end mixes every topic in the styles the slides used in class (trace the output, draw the diagram, define the term, fill in the blank).
- **Glossary** at the very end is a one-screen cram sheet. Read it last, the night before.
- The in-class activities (memory diagram warm-ups, DOM tree drawing, async output prediction) are the strongest hints about question style. All of them are worked in full here.

EXAM TIP. Every in-class attendance check this module was a tracing or drawing exercise done on paper: draw a DOM tree (L04), draw a memory diagram (L05), predict async console output (L06). A paper, closed-book exam is very likely to reuse exactly those formats. Practice them by hand, not just by reading.

L01 1. The Internet

1.1 Course facts worth knowing (from L00/R00)

- Course goal: fundamentals of full-stack web development with emphasis on modern tooling, best practices, and industry-prevalent frameworks. Six modules: Fundamentals (~4 wks), React (~2), Next.js (~2), Backend (~4), Design (~3), More topics (~2). Content is cumulative.
- Grade breakdown: Attendance/Participation 10%, Readings 10%, Coding Assignments 30%, Midterms 20% (two, 10% each), Final Project 30%. The final project presentation replaces a final exam.
- Three free absences; after that absences count unless university-approved. Assignments and readings due 11:59pm; 48-hour late window with a 15% penalty; three late penalties dropped (drops do not apply to zeros).
- Regrade requests: within 48 hours of grade release, one request per question on Gradescope.
- AI policy: AI is allowed for learning, but you may not submit copy-pasted AI code. You must be able to explain every line, and you must cite StackOverflow, ChatGPT, Copilot, etc. in comments. Staff may call you in for a live code review.
- Materials on the course website; submissions on Gradescope; announcements and final grades on Canvas; GitHub account required.

1.2 The Web, the Internet, and networks

Network.
A connection between two or more computers. "Computer" here means any electronic device that can connect to the internet.

The Internet.
The interconnected network of computers around the world. It is infrastructure.

World Wide Web (WWW).
An information system enabling content sharing *over* the internet. The Web runs on top of the Internet; they are not the same thing.

> Watch out. "Internet" and "Web" are not synonyms. The Internet is the network of networks; the Web is one service (content sharing via HTTP) built on top of it. Email, video calls, and games also use the Internet without being "the Web."

1.3 Identifying devices: IP addresses

Devices on a network need a way to be identified so that traffic can be routed to them. That identifier is the **IP address**. The slides draw them as four dot-separated numbers (for example 17.253.144.10 for Apple's server). Every device that participates in a conversation on the internet, your laptop as well as the server, has one.

Extra context beyond the slides: IPv4 addresses are four numbers from 0 to 255 (32 bits). Several examples on the slides, such as 543.33.23.671, are deliberately made-up placeholders and would not be valid IPv4 addresses. 127.0.0.1 is the loopback address, "localhost," which you meet again in L03.

1.4 Protocols and layering

PROTOCOL. A set of rules that define exactly how a communication service is implemented. Also phrased on the slides as: a predefined, universal set of rules that computers use to talk to each other. Requests must be formatted in a way both computers understand.

Protocol layering: complex services are defined on top of simpler services. HTTP (the Web) is built on TCP, which is built on IP, which is built on the physical links. Each layer only needs to know the interface of the layer below it.

TCP / IP. The two protocols at the heart of the Internet. Together they provide a **process-to-process, full duplex, byte stream**. Unpack that phrase:

- *process-to-process*: the connection is between two programs (a browser process and a web server process), not just two machines. IP gets packets to the machine; TCP (with port numbers) gets them to the right process.
- *full duplex*: both sides can send at the same time.
- *byte stream*: the application sees an ordered, reliable stream of bytes, not individual packets.

1.5 Clients, servers, requests, responses

Computers can request resources from other computers on the network. One computer requests a resource; the other responds with that resource. Images, files, data, and websites are all **resources**.

Server.
A computer whose job is to provide resources to other devices on the network.

Client.
A computer that requests resources from a server. Your laptop running a browser is the client.

HTTP (HYPERTEXT TRANSFER PROTOCOL). The protocol used to send information on the web between clients and servers. It consists of a **request** for a resource from a client and a **response** containing the requested resource from the server. The request and response are formatted consistently every time, conforming to the protocol. (The exact format is covered in L05.5.)

HTTPS. A secure version of HTTP that **encrypts the contents of the request and response** so nobody besides the client and server can read it. The "s" in `https://www.apple.com` means the page was requested via HTTPS.

1.6 Anatomy of a URL

When we load a webpage, the client requests the files necessary to render the website from the server tasked with storing them. **URLs tell the browser where to look for a website's files and which ones specifically to look for.**

```
https :// apple.com / iphone
  |          |          |
protocol   domain     path
```

- **Protocol** (`https`): the protocol used to request the website's files.
- **Domain** (`apple.com`): the human-readable name of a website.
- **Path** (`/iphone`): specifies a specific resource, such as a specific page. If the path is omitted, you receive the website's homepage (by convention `index.html`, per R01).

Beyond the slides but appears in L05.5: a URL can also carry a port, as in `http://server:port/path/to/resource`. If no port is given, HTTP defaults to 80 and HTTPS to 443.

1.7 DNS: from domain to IP

Problem: a domain like apple.com identifies a website to humans, but it does not tell the computer which server to contact. **Solution:** the **Domain Name System (DNS)**, "like an address book, but for websites." The browser asks a DNS server for the IP address that corresponds to a domain, and the DNS server replies with it (apple.com → 17.253.144.10). You can try this yourself at nslookup.io.

1.8 All together: what happens when you type a URL

1. User types `https://apple.com` into the browser.
2. The browser (client) requests the IP address for `apple.com` from a **DNS server**.
3. The DNS server replies with the IP of the server holding the website's files (17.253.144.10).
4. The browser requests the files from the server at 17.253.144.10 via HTTPS.
5. Apple's server responds with the files that make up the website: HTML and CSS (and JS, per L03).
6. The browser renders the page from those files.

EXAM TIP. This sequence appeared on three separate decks (L01, L02, L03) as the recurring diagram. Be able to reproduce it in order, name every participant (client, DNS server, web server), say what each message contains, and name the files returned (HTML, CSS, JS).

1.9 Check yourself

1. What is the difference between the Internet and the World Wide Web?

The Internet is the interconnected network of computers around the world. The WWW is an information system for sharing content over the Internet. The Web is a service that runs on top of the Internet.

2. Define "protocol" and "protocol layering."

A protocol is a set of rules that define exactly how a communication service is implemented (a universal set of rules computers use to talk). Layering means complex services are defined on top of simpler services, e.g. HTTP on top of TCP on top of IP.

3. What three properties describe the service TCP/IP provides?

Process-to-process, full duplex, byte stream.

4. Label the parts of `https://comp426-26f.github.io/home`.

Protocol: https. Domain: comp426-26f.github.io. Path: /home. Because a path is present, a specific page is requested rather than the homepage.

5. What does a DNS server return, and why is it needed?

It returns the IP address for a domain name. Domains are human-readable, but computers need the IP address to know which server to contact.

6. What does HTTPS add over HTTP?

Encryption of the request and response contents so that only the client and server can read them.

7. In the client/server model, which is your laptop and which is apple.com's machine? What flows in each direction?

Your laptop is the client (requests resources); Apple's machine is the server (provides resources). The client sends a request; the server sends a response containing the resource.

L02 2. HTML and CSS

2.1 Websites are made of elements

Any website can be broken down into **UI elements**: text, links, images, sections, and so on. The browser needs to know which elements appear on a website and how they are composed so it can render the page. That information comes from HTML.

> HTML (Hypertext Markup Language). The language used to specify the **content and structure** of a webpage. It answers: what elements exist on the webpage?

R01 frames every website as three parts: **content** (HTML), **style** (CSS), and **behavior** (JavaScript). L03 later adds "functionality" as the third piece.

2.2 Tags, content, nesting

```
<html>
  <head>
    <title>Sample website</title>
  </head>
  <body>
    <h1>Hello students!</h1>
    <p>Welcome to COMP 426</p>
    <p>
      We are super excited to have
      you in our class.
    </p>
  </body>
</html>
```

- Elements are described using **tags**. `<element>` starts an element; `</element>` ends it. The text between `<` and `>` is the element's **type** (`p` is a paragraph).
- Everything between the start and end tags is the element's **content**.
- Elements can be **nested**: an element can be the content of another element. The paragraphs above are part of the content of `<body>`.
- HTML comments look like `<!-- comment -->`.

2.3 File structure: html, head, body

All elements are wrapped in an `<html>` element, which must contain two sub-elements:

- `<head>`: contains **metadata** about the webpage, such as the `<title>` (shown in the browser tab) and links to external resources like stylesheets and scripts.
- `<body>`: contains the **content** of the webpage: all elements to be displayed.

R01 adds `<!DOCTYPE html>` as the first line, which tells the browser the document is HTML5, and notes that `<html>` is always the root element.

2.4 Properties (attributes)

Some elements accept **properties**, also called **attributes**, written inside the opening tag, to aid the element's function. Examples: `<img src="url">`, `<a href="url">`. Two attributes are allowed on *all* elements:

- `class=""`: similar elements can share the same class for easy handling as a group. An element can have several classes separated by spaces: `<p class="text post">`.
- `id=""`: **uniquely identifies** an element. One id per element, one element per id.

2.5 Common elements (R01 reference table)

Tag	Purpose
`<h1>` … `<h6>`	Headings from largest (h1) to smallest (h6).
`<p>`	Paragraph of text.
`<div>`	Container that groups elements in a **block**. One of the most important elements for layouts.
`<span>`	Container that groups **inline** content, e.g. to style part of a sentence without a line break.
`<em>` / `<strong>`	Italic / bold text.
` `	Line break.
`<a href="url">`	Link. The content is the visible text; `href` is the destination.
`<img src="url">`	Image from the given source.
`<ul>` / `<ol>` with `<li>`	Unordered (bulleted) / ordered (numbered) list of list items.
`<table>`, `<tr>`, `<th>`, `<td>`	Table, row, header cell, data cell.
`<link rel="stylesheet" href="styles.css">`	In `<head>`: links a CSS file.
`<script src="app.js"></script>`	Links a JavaScript file (L03).

Relative paths (R01): `href="pages/about.html"` looks in a subfolder; `href="../index.html"` goes up one folder. `index.html` is the conventional homepage served when the URL path is `/`.

2.6 Browser developer tools

Every standard browser (Chrome, Safari, Firefox, Edge) has developer tools that let you view and edit the HTML of any page, inspect CSS (including a live box-model view), and see the console. The first attendance check was a screenshot of dev tools. A02 asks for a screenshot of the Network tab, which shows every request the browser makes.

2.7 CSS: style

HTML only defines the content of the website, not how it looks. Without CSS, a page is very plain.

CSS (Cascading Style Sheets). Used to apply styling to HTML elements. R01: a language to specify the **style and layout** of webpages (colors, sizes, fonts, spacing, positions).

```html
<!-- index.html -->
<html>
  <head>
    <title>Sample website</title>
    <link rel="stylesheet" href="styles.css">
  </head>
  <body>
    <h1 id="header">Hello students!</h1>
    <p class="text">Welcome to COMP 426</p>
    <p class="text post">We are super excited...</p>
    <img src="url">
  </body>
</html>
```

```css
/* styles.css */
#header {
  color: blue;
}
.text {
  color: red;
  letter-spacing: 20px;
}
.post {
  font-style: italic;
}
img {
  border-radius: 50%;
}
```

- The `<link rel="stylesheet" href="styles.css">` element in `<head>` links a CSS file to the HTML. (You can also write CSS inside a `<style>` tag in the head, but separate files are best practice.)
- A CSS **rule** = a **selector** (which elements) + a block of **declarations** in `{ }`, each `property: value;`.
- `#header` selects the element with **id** "header". `.text` selects all elements with **class** "text". `img` selects all elements of **type** img.
- The second paragraph has both classes, so it gets red, letter-spaced, and italic. That is the "cascading": multiple rules combine on one element.

2.8 More selector patterns

Pattern	Meaning	Example
`.class`, `#id`, `element`	Simple selectors: by class, by id, by element type.	`.post`, `#header`, `p`
`E.class`, `E#id`	Class/id selector limited to elements of type E.	`p.text` = paragraphs with class text (not h2s with that class).

`E F`	Descendant: F elements nested anywhere inside an E.	`div p` = every p inside a div at any depth.
`E > F`	Child: F elements that are **direct** children of E.	`ul > li`
`E + F`	Adjacent sibling: an F that **immediately follows** an E (same parent).	`h1 + p` = the paragraph right after a heading.
`E:hover`, `E:first-child`, `E:focus`, …	Pseudo-classes: match an element in a particular state or position.	`a:hover`

Compound selectors chain these together and are **read from right to left**. The slide's example:

```
div.important p + p + p span:hover
```

Read as: applies to **span** elements that are being **hovered** over, that are **descendants of a p** element, which is an **immediate sibling of a p**, which is an **immediate sibling of a p**, which is a **descendant of a div** whose class attribute includes "important." So: hovering a span inside the third-or-later of three consecutive paragraphs inside an important div.

EXAM TIP. A very plausible question: "Which elements does this selector match?" or "Write a selector for …". Practice reading right to left, and keep `E F` (any descendant) and `E > F` (direct child only) straight.

2.9 Layout: what happens without CSS

Disabling CSS on amazon.com (with the "Web Developer" Chrome extension) does not just change fonts and colors: crucially, it changes the **layout**. By default, without CSS, all items get arranged **top to bottom**. CSS lets us manipulate how items are arranged. Making good layouts is one of the hardest CSS skills but the most important one.

Layout viewport.
The area on which the browser arranges the elements of a webpage. It has a width and height, measured in pixels (px), determined by the device loading the page.

Default placement.
Items start at the top-left corner of the layout viewport, then stack below. Viewing the screen as a coordinate system, the origin (0,0) is the top-left, with x increasing to the right and y increasing **downward**. Each element's position is described by its top-left corner.

2.10 The CSS box model

The box model describes how HTML elements are positioned relative to nearby elements, how an element's contents are positioned within it, and the properties of its border. From the outside in:

Layer	Definition	CSS

Margin	Space **outside** the border of the element (separates it from neighbors).	`margin`, `margin-top`, …
Border	The visible border around an element's content, if any.	`border: 2px dashed red;`
Padding	Space between an element's border and its content.	`padding`, `padding-right`, …
Content	Where the element's content is displayed.	`width`, `height`

```
#my-element {
  margin: 20px 10px 20px 10px;   /* top right bottom left, or: */
  margin-top: 36px;
  border: 2px dashed red;
  padding: 20px 10px 20px 10px;  /* or: */
  padding-right: 10px;
  width: 300px;
  height: 400px;
}
```

Watch out: width and height only size the content box.. The element's total width is

margin-left + margin-right + 2 × border width + padding-left + padding-right + content width.

For the rule above, total width = 10 + 10 + 2 · 2 + 10 + 10 + 300 = 344px (using the four-value `margin` shorthand; the `margin-top: 36px` override only affects height). Total height = 36 + 20 + 4 + 20 + 20 + 400 = 500px. Be ready to compute these.

Four-value shorthand order is clockwise from the top: top, right, bottom, left. Nested elements sit inside the *content* area of their parent, which is why the parent's padding pushes children inward.

2.11 Display modes and flexbox

Elements are placed top to bottom by default. To put items next to each other we change an element's **display mode**. Options include `block`, `inline`, `grid`, and `flex`. The course focuses on the most powerful one for complex layouts: flexbox.

CSS Flexbox (Flexible box layout). A layout method allowing items to be arranged in **rows and columns**. One of the most powerful and versatile layout methods: simple building blocks capable of creating some of the most complex layouts.

- **Flex container**: an element with the flex display mode enabled (`display: flex;`). Its children become flex items.
- `flex-direction` determines how the container's content is arranged: `row` (side by side, left to right) or `column` (stacked top to bottom).

```html
<body>
  <div id="flex-container">
    <h1 id="one">Hark</h1>
    <p class="two">the sound.</p>
  </div>
</body>
```

```css
#flex-container {
  display: flex;
  flex-direction: row;
}
/* "Hark" and "the sound." now sit
   side by side instead of stacked */
```

Supplement S01 (flexbox practice) and A01 go further with `justify-content`, `align-items`, `gap`, and `flex-wrap`, which distribute and align items along the main and cross axes. Those were not on the L02 slides but are fair background.

2.12 Does HTML or CSS determine layout? Both.

The lecture's closing "open question," answered on the next slide: **BOTH**.

- Without CSS, the box model (margin, padding, etc.) would not exist.
- Without HTML, there would be no composition relationship between elements. That matters for padding and margin, since nested elements are considered inside the content of their parent element.
- Without HTML, flexboxes would not be possible (a flex container needs children to arrange).

Recap: HTML provides structure and content, CSS provides styling, and **HTML + CSS together provide layout**. CSS Zen Garden (csszengarden.com) shows one HTML file restyled into radically different sites purely via CSS. The answer to "what files make up a website?" is HTML and CSS (and, from L03, JavaScript).

2.13 Check yourself

1. What goes in `<head>` versus `<body>`?

> `<head>` holds metadata: the title and links to external resources (stylesheets, scripts). `<body>` holds the content that is displayed: all visible elements.

2. Difference between `class` and `id`?

> `id` uniquely identifies one element (selected with `#id`). `class` groups similar elements so they can be handled together (selected with `.class`); an element can have several classes and many elements can share one class.

3. Which elements does `ul > li a:hover` match?

> Anchor elements being hovered that are descendants of an `li` that is a direct child of a `ul`.

4. An element has `width: 200px; padding: 10px; border: 5px solid; margin: 15px`. What is its total width?

> $15 + 15 + 5 + 5 + 10 + 10 + 200 = 260$px.

5. What is the layout viewport, and where is (0,0)?

The area the browser arranges elements on; width and height in pixels determined by the device. The origin is the top-left corner, with x increasing right and y increasing down. Each element's position is described by its top-left corner.

6. What does `display: flex` do to an element, and what does `flex-direction: column` mean?

It makes the element a flex container whose children are arranged in rows or columns. `column` stacks the children vertically; `row` lays them out horizontally.

7. Why is the answer to "does HTML or CSS determine layout" both?

CSS supplies the box model and display modes such as flex; HTML supplies the nesting (composition) that padding, margin, and flex containers operate on. Neither alone yields the layout.

8. Write the HTML to link `styles.css` and the CSS to make every paragraph with class "note" italic.

`<link rel="stylesheet" href="styles.css">` in the head; `p.note { font-style: italic; }` (or `.note { ... }` if any element with the class should match).

L03 3. JavaScript, TypeScript, and Build Systems

3.1 The missing piece: functionality

HTML provides structure and content. CSS provides styling. HTML + CSS together provide layout. The missing piece is **functionality**, provided by JavaScript (and in this course, TypeScript compiled to JavaScript).

Traditionally you link a script exactly like a stylesheet, from the `<head>`:

```
<html>
  <head>
    <title>Sample website</title>
    <script src="app.js"></script>
    <link rel="stylesheet" href="styles.css">
  </head>
  <body>...</body>
</html>
```

3.2 How JS loads (the request sequence)

1. The client requests the website's home page (path `/`).
2. The server responds with `index.html`.
3. The browser **parses the HTML and builds the element tree, a.k.a. the DOM**.
4. While parsing `<head>`, the browser encounters the `<link>` and `<script>` tags and **requests the CSS and JS files**. The server responds with them.
5. The browser renders the page according to the CSS.
6. The browser **runs each JavaScript file as it is encountered**.

EXAM TIP. Note the key phrase: requests for CSS/JS are made *when the head is parsed*, and JS runs *as encountered*. This is the same client/server diagram from L01 extended to three files, and it ties into L05.5's "typical use pattern": one exchange for the HTML, then subsequent exchanges for stylesheets, scripts, and images.

3.3 TypeScript is a superset of JavaScript

TypeScript extends JavaScript: every JS program is valid TS, and TS adds features that make development better. The headline feature:

Static vs dynamic typing.

- **Dynamically typed (JavaScript):** variables' data types are determined at **runtime**. You can assign any type of value to a variable at any point.
- **Statically typed (TypeScript):** variables' data types are declared or inferred at **compile time**. The compiler can type-check and catch errors early, before runtime.

```javascript
// script.js (dynamic)
function add(a, b) {
  return a + b;
}
console.log(add(2, 3));      // 5
console.log(add("2", "3"));  // "23"  (silently concatenates)

const user = { name: "Ramses", age: 28 };
console.log(user.address);   // undefined (no error!)
```

```typescript
// script.ts (static)
function add(a: number, b: number): number {
  return a + b;
}
console.log(add(2, 3));      // 5
console.log(add("2", "3"));  // COMPILE-TIME ERROR

const user = { name: "Ramses", age: 28 };
console.log(user.address);   // COMPILE-TIME ERROR
```

Why the course uses TypeScript: static typing (and many other features) reduces errors significantly, roughly 20% of all JS errors; it is helpful in the classroom; it is used heavily in industry (the slides cite a developer survey). If you learn TS, you learn JS as well.

"Type inference" means TS figures out a type from the value when you do not annotate: `let x = 5` makes `x` a `number`, and later assigning a string to it is a compile error.

3.4 The problem: browsers only understand JavaScript

- The browser only understands JavaScript. JavaScript is an **interpreted** language.
- TypeScript code has to be **compiled into JavaScript** so the browser can understand it. Because the source and target are both high-level languages, this is called **transpilation**.
- Transpilation happens when we **build** our project. Building gets the project ready and configured to be deployed.
- If you tried to serve `app.ts` directly, the browser would not be able to run it.

Workflow: (1) develop the project in VS Code (HTML, TS, CSS); (2) build the project, during which TS is transpiled to JS; (3) put the output on the server; (4) the client now receives a JS file instead of TS.

Course	Source	Tool	Output / runs on
COMP 211	`main.c`	gcc	`a.out` executable run by the shell
COMP 210/301	`Main.java`	Maven or Gradle (build) + javac	`Main.class` bytecode run on the JVM
COMP 426	`main.ts`	Vite (build) + tsc	`main.js` run in the browser

3.5 Build tools: Vite

Many build tools convert TS to JS. The course uses **Vite** (pronounced "veet"): a very powerful and fast build tool. StackBlitz uses Vite by default. Vite also simulates a server we can access locally (the dev server). Vite is an example of an **external package**.

3.6 Environments, Node.js, and npm

Vite itself is written largely in TypeScript/JavaScript. Typically JS runs inside a browser, but browser JS cannot access computer files, the terminal, or run shell processes. To use a build tool like Vite, we need to run JavaScript **outside the browser**, directly on our machine, so it can see the project's source code and generate JS files.

Node.js.
A **JavaScript runtime environment** that allows us to run JavaScript code outside of a browser.

Environment.
A space where code can be run. It can contain other **packages** (external JS code written by others) that we can run directly, or that our code can reference and use as a **dependency**. Packages can depend on other packages. The Node environment sits on your local computer.

npm (Node Package Manager).
Used to install packages into the Node environment: `npm install <package>`. Analogy from COMP 110: `pip install <package>` installs into your local Python environment; pip is Python's package manager.

3.7 The dev server: `npm run dev`

- `npm run dev` starts a **development server**: a "fake"/simulated server that runs locally on your computer rather than on the internet.
- Vite builds the project and puts the output HTML/CSS/JS on this server for you to interact with.
- It is accessible at IP **127.0.0.1**, a.k.a. **localhost**, on a certain **port**. Ports allow multiple servers to run locally at once. Example: `http://localhost:3000`. (A02's Vite config uses port 4260.)

L03 SUMMARY SLIDE, NEARLY VERBATIM. In COMP 426 we use TypeScript to add functionality to webpages. Browsers only recognize JavaScript, so we transpile TS → JS. This is worth it because TS is statically typed. Vite is a build tool that handles the process. To make this happen we put our code in a Node environment with packages (installed with npm) like Vite. `npm run dev` builds the project and spins up a local dev server at `localhost:3000`.

3.8 Check yourself

1. In what order do a browser's requests happen when loading a page with a linked stylesheet and script, and when does the JS run?

Request `/` → receive index.html → parse HTML and build the DOM → while parsing head, request CSS and JS → receive them → render per CSS → run each JS file as encountered.

2. Define static and dynamic typing and give the JS vs TS behavior for `add("2","3")` when `add` was written for numbers.

Dynamic: types determined at runtime, any value can be assigned to any variable (JS). Static: types declared or inferred at compile time so the compiler can catch errors early (TS). JS returns "23" (string concatenation); TS with `a: number, b: number` gives a compile-time error.

3. Why must TypeScript be transpiled, and when does that happen?

Browsers only understand JavaScript. TS is transpiled to JS during the build step, which prepares the project for deployment.

4. What is Node.js and why do we need it to use Vite?

A JavaScript runtime that runs JS outside the browser. Vite is written in JS/TS and needs file and shell access that browser JS lacks, so it runs in Node on your machine.

5. What does npm do, and what is the Python analogue?

Installs packages into the Node environment (`npm install <pkg>`). Python's analogue is pip.

6. What is localhost, and why do we need a port number?

localhost is the IP 127.0.0.1, your own machine. The port lets several local servers run at once, e.g. `localhost:3000`.

7. Match: gcc, Maven, Vite to the course and the artifact produced.

gcc (COMP 211) → a.out executable; Maven/Gradle (COMP 210/301) → .class bytecode for the JVM; Vite (COMP 426) → .js for the browser.

L04 4. Event Handling and the DOM

4.1 User interactions as events

To add functionality we need to **respond to events**, which describe users' interactions with the page. Examples from the slides: clicking on an element, pressing a key, dragging an element across the screen, the mouse entering a region, submitting a form, and many more (the slide notes link to the W3Schools DOM event reference).

Key idea: we often want to run some TS code when a user interacts with an element, e.g. alert the user when a button is pressed. Elements are defined in the HTML, so first we need a way to **reference HTML elements from TS code**. That is what the DOM gives us.

4.2 The Document Object Model

DOM (Document Object Model). A data representation of the elements of a webpage. The state of the content and structure of the webpage is represented as a **tree**. Elements defined in the HTML are accessible as **objects** from TS code. (L03: the browser builds this tree when it parses the HTML.)

Rules for drawing a DOM tree

1. The tree **always starts with a Document node**.
2. The root element `<html>` has a special label: **documentElement**.
3. All child elements of an element are placed as **children** in the tree (in order).
4. **Text** is represented with its own node and is a child of the element it appears in.
5. **Attributes** of an element are nodes **associated with** the element but are **NOT children**. Draw them hanging off the side of the element, labelled `Attribute: name Value: value`.

Slide example:

```
<!DOCTYPE html>
<html>
  ...
  <body>
    <div>
      <p>Hi!</p>
    </div>
    <div>
      <a href="url">Link</a>
    </div>
  </body>
</html>
```

```
Document
└── documentElement <html>
    └── Element: <body>
        ├── Element: <div>
        │   └── Element: <p>
        │       └── Text: "Hi!"
        └── Element: <div>
            └── Element: <a>  ~~ Attribute: href, Value: "url"
                └── Text: "Link"
```

The in-class attendance check (worked here in full):

```
<!DOCTYPE html>
<html>
  ...
  <body>
    <h1>Welcome to COMP 426!</h1>
    <p id="text">Hello, world!</p>
    <div class="row">
      <p>a01</p>
      <p>a02</p>
    </div>
  </body>
</html>
```

```
Document
└── documentElement <html>
    └── Element: <body>
        ├── Element: <h1>
        │   └── Text: "Welcome to COMP 426!"
        ├── Element: <p>  ~~ Attribute: id, Value: "text"
        │   └── Text: "Hello, world!"
        └── Element: <div>  ~~ Attribute: class, Value: "row"
            ├── Element: <p>
            │   └── Text: "a01"
            └── Element: <p>
                └── Text: "a02"
```

WATCH OUT. Three classic mistakes: forgetting the Document node at the top, drawing attributes as children, and forgetting that text is its own node (an element like `<p>Hi!</p>` has a child, the text node, not "content inside the box"). Also keep children in document order. If the slide shows `<head>` under "…", you may draw it as a sibling of body when the full HTML is given.

4.3 Accessing the DOM in TypeScript

In a TS file running in the browser, the global **`document`** object stores the DOM. It is a global variable provided by the browser JavaScript environment (it does not exist in Node). Through it we access and directly manipulate elements:

```
const header = document.getElementById("header");
```

Searching / matching elements

Method	Returns
`document.getElementById("id")`	The single node with that id attribute, or `null` if none.
`document.getElementsByTagName("p")`	A node list of elements of that type.
`document.querySelector("css selector")`	The **first** node matching a CSS selector (any selector from L02). Can be called on a node to limit the search to the subtree below it.
`document.querySelectorAll("css selector")`	**All** nodes matching a CSS selector. Also callable on a node to search only its subtree.

Because `getElementById` can return `null`, TypeScript will make you handle that (for example `header!` or an `if (header)` check) before you use the element. `querySelector` accepts the same selectors as CSS, e.g. `document.querySelector("#canvas .cell")`.

Navigating the DOM

Text and comment nodes are interspersed within elements (including whitespace text nodes between tags). Often you just want to navigate the **element** tree, so there are two parallel sets of properties:

Property	Core node tree (all nodes)	Element-only tree
Parent	`parentNode`	`parentElement`
Children list	`childNodes`	`children`
First child	`firstChild`	`firstElementChild`
Last child	`lastChild`	`lastElementChild`
Next sibling	`nextSibling`	`nextElementSibling`
Previous sibling	`previousSibling`	`previousElementSibling`

The API documentation is on MDN: developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model.

4.4 Handling events: addEventListener

```
element.addEventListener("click", () => {
  // code here runs when the element is clicked
});
```

- First argument: the **text name of the event** that should occur for the function to execute (`"click"`, `"keydown"`, `"mouseenter"`, `"submit"`, …).
- Second argument: the **function** (arrow syntax) that should run when the event occurs. This is the **event handler**.
- `addEventListener` is an **example of a higher-order function**: it takes a function as a parameter. (L05 and L06 both return to this: handlers are functions run in response to events, and they are a kind of callback.)

Combining DOM manipulation and event handlers is powerful: add, change, or remove elements in the DOM when an event is triggered.

4.5 Reading and writing content

Property	Meaning
`innerHTML`	Get/set the content of a node **as HTML**. Setting `el.innerHTML = "<b>hi</b>"` creates a bold element.
`innerText`	Get/set the content of a node **as text**, not interpreted as HTML. Setting it to `"<b>hi</b>"` shows the literal characters.

Creating and inserting content (the L06 practice hints and the L04 "inserting content relative to an existing element" demo):

```
const div = document.createElement("div");   // make a new element (not yet in the tree)
const p = document.createElement("p");
p.innerText = "Hello";
div.append(p);                                // insert as last child of div
document.body.append(div);                    // now it is in the DOM and visible

// Relative insertion helpers on any element:
el.append(node)   // as last child        el.prepend(node)  // as first child
el.before(node)   // as previous sibling  el.after(node)    // as next sibling
el.remove()       // remove el from the tree
```

4.6 The event object

Additional information about the event is provided in the **first parameter** to the handler, known as the **event object**. Its properties depend on the kind of event (a keyboard event has `key`, a mouse event has `clientX`/`clientY`), but some properties are universal to all events:

Property	Meaning (slide wording)
`type`	A string with the name of the event, like "click" or "keydown".
`target`	The specific DOM element where the event **first happened**.
`currentTarget`	The element that **currently has the event listener attached** and running.
`bubbles`	true/false: whether the event moves up (bubbles) through parent elements.
`cancelable`	true/false: whether you can stop the default action of the event.
`isTrusted`	true/false: whether a real user action caused the event (vs. JavaScript dispatching it).
`timeStamp`	The exact time in milliseconds when the event was created.

```
grid.addEventListener("click", (event) => {
  // event.currentTarget === grid (where the listener is)
  // event.target is the specific cell that was clicked (it bubbled up to grid)
  const cell = event.target as HTMLElement;
  cell.style.backgroundColor = "red";
});
```

`target` vs `currentTarget` is the classic distinction: click a `<p>` inside a `<div>` that has the listener, and `target` is the p while `currentTarget` is the div. Because events bubble, one listener on a parent can handle clicks on all of its children (this is exactly how A02's pixel grid can work). `event.preventDefault()` uses the `cancelable` property to stop, for example, a form submission from reloading the page.

4.7 Check yourself

1. Define the DOM in one sentence.

A data representation of a webpage's elements, in which the content and structure are represented as a tree and elements are accessible as objects from TS code.

2. Draw the DOM tree for `<body><ul id="list"><li>One</li><li><em>Two</em></li></ul></body>`.

```
Document
└── documentElement <html>
    └── Element: <body>
        └── Element: <ul>  ~~ Attribute: id, Value: "list"
            ├── Element: <li>
            │   └── Text: "One"
            └── Element: <li>
                └── Element: <em>
                    └── Text: "Two"
```

3. What is the difference between `querySelector` and `querySelectorAll`, and between `getElementById` and `querySelector("#x")`?

`querySelector` returns the first match; `querySelectorAll` returns all matches. `getElementById("x")` and `querySelector("#x")` both return the element with id x (or null); the latter accepts any CSS selector and can be scoped to a subtree.

4. Why is `addEventListener` a higher-order function?

It accepts a function (the handler) as a parameter.

5. `childNodes` vs `children`?

`childNodes` includes every node (text, comments, elements); `children` includes only element nodes.

6. innerHTML vs innerText?

innerHTML gets/sets content as HTML (tags are interpreted); innerText gets/sets it as plain text (tags shown literally).

7. A click listener is on a `<div>`; the user clicks a `<button>` inside it. What are `event.target` and `event.currentTarget`?

`target` is the button (where the event first happened); `currentTarget` is the div (where the listener is attached).

8. Write TS that finds the element with id "count" and, when a button with id "inc" is clicked, increments the number shown.

```ts
const count = document.getElementById("count")!;
const btn = document.getElementById("inc")!;
let n = 0;
btn.addEventListener("click", () => {
  n += 1;
  count.innerText = String(n);
});
```

(Note `n` lives in a closure; see L05.)

L05 5. Functional Programming, Memory Diagrams, and Closures

5.1 The instructor's memory diagram notation

This lecture is almost entirely worked memory diagrams, and both in-class attendance checks were "draw the diagram." Learn the notation exactly.

Stack (left column): one frame per active call

Globals (F0)	P: -
num	426
halve	id:0

halve (F1)	P: F0
RA	8
RV	213
x	426

- **Frame label**: function name and frame number in creation order. Globals is always **F0**.
- **P (parent frame)**: the frame in which the called function was *defined*, copied from the heap function object's P. This is lexical scoping, not who called it.
- **RA (return address)**: the line number execution returns to when the frame finishes.
- **RV (return value)**: filled in when `return` executes.
- Then one row per parameter and local variable.
- Frames are pushed on call and **popped on return**; cross out popped frames.

Heap (right column): functions, objects, strings

id:0	P: F0
fn	lines 4-6
x: number	

id:4	(object)
name	id:5
greet	id:6

id:3	string
"A J"	

- Every **function definition** goes in the heap and gets an **id**. Record its **P** (the frame it was defined in), the **lines** of its body, and its parameter list.
- A variable holding a function stores the **id**, not the code.
- **Object literals** are heap objects with one row per field; field values that are functions or strings hold ids.
- **Strings** are heap values with ids too (used in the Person example).
- Numbers are stored directly in the frame.

5.2 Worked example 1: a single call

```
1  const num = 426;
2
3  const halve =
4    (x: number) => {
5      return x / 2;
6    }
7
8  const result = halve(num);
```

1. **Line 1:** `num: 426` is placed in the Globals (F0) frame.
2. **Lines 3–6:** the function definition is placed in the heap as **id:0** with **P: F0** (defined inside globals), **fn lines 4–6**, param `x: number`. The variable `halve` in F0 gets the value **id:0**.
3. **Line 8:** calling `halve(num)` needs a **new stack frame**: `halve (F1)`, **P: F0** (from id:0's P), **RA: 8** (we return to line 8 to finish the assignment), `x: 426`.
4. **Line 5:** `return x / 2` sets **RV: 213**. Execution returns to the line stored in RA.
5. Back in F0, `result: 213` is recorded. F1 is **popped off the stack**. Done.

5.3 Worked example 2: nested calls (attendance warm-up)

```
1  const x = 12;
2
3  const multiply =
4    (a: number, b: number) => {
5      return a * b;
6    }
7
8  const square = (a: number) => {
9    return multiply(a, a);
10 }
11
12 const result = square(x);
```

Final state of the heap: **id:0** (P: F0, fn lines 4–6, params a, b) and **id:1** (P: F0, fn line 9, param a). Globals F0: `x: 12`, `multiply: id0`, `square: id1`, `result: 144`.

Sequence of frames:

square (F1)	P: F0
RA	12
RV	144
a	12

multiply (F2)	P: F0
RA	9
RV	144
a	12
b	12

- F1 is created for `square(x)` with RA 12 and `a: 12`. Its P is F0 because `square` was defined in globals.
- Line 9 calls `multiply(a, a)`, creating F2 with **RA 9**, `a: 12`, `b: 12`. Its P is also **F0** (where `multiply` was defined), *not* F1 (the caller). Parent frames are about definition, not calling.
- F2 returns 144 (RV) → popped. Line 9 returns that value, so F1's RV is 144 → popped. `result: 144` in F0.

WATCH OUT. The single most common memory-diagram error is setting P to the calling frame. P is *always* the frame in which the function was *defined* (the P recorded on the heap function object). This is what makes closures work in 5.6.

5.4 Worked example 3: the "mystery" higher-order function

```
const mystery = (nums: number[], fn: (x: number) => number): number[] => {
  result = [];
  for (const num of nums) {
    result.push(fn(num));
  }
  return result;
}
const operationFactory = (factor: number) => {
  return (x: number) => x * factor;
}
const operation = operationFactory(3);
const result = mystery([1, 2, 3], operation);
```

Intended result: `[3, 6, 9]`. `mystery` is a hand-written `map`: it applies `fn` to every element and collects the outputs. `operationFactory(3)` returns a function that multiplies its input by 3 (the returned function *closes over* `factor`). Passing that to `mystery` triples each element.

Nit: as written, `result = [];` inside `mystery` has no `let`/`const`, so it would refer to the global `const result`, which TypeScript rejects. Assume the slide meant `const result = [];` inside the function. If an exam question has an obvious typo like this, answer the intended question and briefly note the issue.

5.5 Higher-order functions

HIGHER-ORDER FUNCTION. A function that either **returns another function**, or **accepts a function as a parameter** (or both). Higher-order functions unlock an entirely new paradigm of programming.

From R03 (see section 8 for the full syntax review): functions are **values**. Arrow syntax creates a **function literal**, an anonymous function value you can store in a variable, pass as an argument, or return. A function's type is written from its parameters and return type: `(num: number) => number`. Type aliases make these readable: `type NumberTransformer = (num: number) => number;`.

```
// Function as a parameter
const mapNumbers = (
  nums: number[],
  transform: (num: number) => number
): number[] => {
  const out: number[] = [];
  for (const n of nums) out.push(transform(n));
  return out;
};
mapNumbers([0, 1, 4], (n) => n * 2); // [0, 2, 8]
```

```
// Function as a return value (a "factory")
const makeMultiplier =
  (factor: number): ((num: number) => number) => {
    return (num: number): number => num * factor;
  };
const triple = makeMultiplier(3);
triple(5);                          // 15
mapNumbers([0, 1, 4], triple);   // [0, 3, 12]
```

Examples you have already used: `addEventListener(type, handler)` (L04) takes a function; `.then(callback)` (L06) takes a function; `operationFactory` returns a function.

5.6 Closures: the counter example

```
1  const updater = () => {
2    let counter = 0;
3
4    return () => {
5      counter += 1;
6      console.log(counter);
7    }
8  }
9
10 const update = updater();
11
12 update();   // prints 1
13 update();   // prints 2
14 update();   // prints 3
```

This runs without errors and prints **1, 2, 3**. It is surprising because `counter` is a local variable of `updater`, whose call has already returned, yet the inner function still reads and updates it. The inner function **references a value outside its immediate scope but still has access to it**. This is called a **closure**.

Memory diagram

STACK

Globals (F0)	P: -
updater	id:0
update	id:1

updater (F1)	P: F0
RA	10
RV	id:1
counter	0 → 1 → 2 → 3

update (F2)	P: F1
RA	12
RV	-

HEAP

id:0	P: F0
fn	lines 1-8

id:1	P: F1
fn	lines 4-7

1. `updater` is defined: heap **id:0, P: F0**. F0 gets `updater: id0`.
2. Line 10 calls `updater()`: frame **F1** (P: F0, RA 10) with local `counter: 0`.
3. Lines 4–7 define the inner arrow function *while F1 is active*: heap **id:1 with P: F1**. F1 returns it: RV id:1. F0 gets `update: id1`.
4. Line 12 calls `update()`: frame **F2 with P: F1** (copied from id:1). Since F2's parent frame is F1, **F2 has access to the variables stored in F1**, including `counter`. It increments it to 1 and logs 1.
5. Lines 13 and 14 each create a new frame (F3, F4) with P: F1, and `counter` in F1 keeps its updated value: 2, then 3.

CLOSURE. A function bundled with the environment (frame) in which it was defined. Even after the defining function has returned, the inner function keeps access to that frame's variables through its P link, so state persists between calls. The F1 frame is *not* garbage: it stays alive because a live heap function (id:1) points to it.

EXAM TIP. Expect a "does this run, and what does it print?" question like this one, possibly with two independent counters (`const a = updater(); const b = updater();` each get their own F1-style frame, so they count separately), or a factory like `operationFactory`. Always ask: where was the inner function *defined*? That frame is its P.

5.7 Functional programming as a paradigm

- **Programming paradigms** are ways of structuring and organizing code. Object-oriented programming (COMP 110 onward) is one paradigm, but not the only one.

- **Functional programming:** instead of using classes, all problems can ultimately be modelled using functions.
- None of this would be possible without closures. In older JavaScript there was no `class` notation; objects were created as literals or purely with functions, and **encapsulation (private fields) was only possible with closures**.
- Web programming relies heavily on functional programming: event handlers are functions running in response to events, and starting in the React module you will see core parts of web apps expressed as, or composed of, functions. You have seen it in Python (COMP 110) and Java lambda expressions (COMP 301).

5.8 Objects from closures: the Person example

```
// Object-oriented version
class Person {
  name: string;
  constructor(fn: string, ln: string) {
    this.name = fn + " " + ln;
  }
  greet() {
    return "Hi, " + this.name + "!";
  }
}
const aj = new Person("A", "J");
console.log(aj.greet());   // Hi, A J!
```

```
// Functional version (closure + object literal)
const Person =
  (fn: string, ln: string) => {
    let name = fn + " " + ln;
    return {
      name: () => name,
      greet: () => {
        return "Hi, " + name + "!";
      }
    }
  }
const aj = Person("A", "J");
console.log(aj.greet());   // Hi, A J!
```

- `{ ... }` with fields is an **object literal**: we can create objects without a class.
- `name` is a local of the `Person` call. The two arrow functions close over it. Nothing outside can touch `name` directly: that is **encapsulation via closure**, the functional equivalent of a private field.
- Because `name` is only exposed through the `name()` function, it is not mutable from outside. JavaScript offers **getter/setter syntax** that looks like a directly accessible field:

```
return {
  get name() { return name; },
  set name(n: string) { name = n; },
  greet: () => "Hi, " + name + "!"
}
const aj = Person("A", "J");
aj.name = "XYZ";           // calls the setter
console.log(aj.greet());   // Hi, XYZ!
```

Memory diagram for the functional Person (the 10-minute in-class exercise)

```
1  const Person =
2    (fn: string, ln: string) => {
3      let name = fn + " " + ln;
4      return {
5        name: () => name,
6        greet: () => {
7          return "Hi, " + name + "!";
8        }
9      }
10   }
11
12 const aj = Person("A", "J");
13 const greeting = aj.greet();
14 console.log(greeting);
```

Stack

Globals (F0)	**P: -**
Person	id:0
aj	id:4
greeting	id:7

Person (F1)	**P: F0**
RA	12
RV	id:4
fn	id:1
ln	id:2
name	id:3

greet (F2)	**P: F1**
RA	13
RV	id:7

Heap

id:0	**P: F0**
fn lines 2-10	fn: string, ln: string

id:1	**string**
"A"	

id:2	**string**
"J"	

id:3	**string**
"A J"	

id:4	**object**
name	id:5
greet	id:6

id:5	**P: F1**
fn line 5	

id:6	**P: F1**
fn lines 6-8	

id:7	string
"Hi, A J!"	

Walkthrough: defining `Person` creates id:0. Calling it (line 12) creates F1 with RA 12; the string arguments are heap strings id:1 and id:2; line 3 builds "A J" as id:3. The object literal becomes id:4 with two fields whose values are the two inner functions id:5 and id:6, both with **P: F1**. F1 returns id:4, stored in `aj`. Line 13 calls `aj.greet()`: F2 has **P: F1**, so it can read `name` (id:3) and build "Hi, A J!" as id:7, returned into `greeting`. Console prints `Hi, A J!`.

5.9 Check yourself

1. In a stack frame, what do P, RA, and RV stand for, and how is P determined?

P = parent frame (the frame where the function was defined, taken from the heap function object), RA = return address (line to resume at), RV = return value. P is about lexical definition, not the caller.

2. Trace and give the output.

```
const makeAdder = (n: number) => (x: number) => x + n;
const add2 = makeAdder(2);
const add10 = makeAdder(10);
console.log(add2(5), add10(5), makeAdder(1)(1));
```

`7 15 2`. Each `makeAdder` call gets its own frame holding `n`; each returned function's P points at its own frame.

3. Does this run, and what does it print?

```
const counterA = updater();   // updater from 5.6
const counterB = updater();
counterA(); counterA(); counterB();
```

Yes. Prints 1, 2, 1. Each call to `updater()` creates a separate frame with its own `counter`, so the two closures are independent.

4. What is a higher-order function? Name two you have used in this course.

A function that takes a function as a parameter and/or returns a function. Examples: `addEventListener`, `.then`, `mapNumbers`, `operationFactory`, `Array.map`.

5. Why did pre-class JavaScript need closures for encapsulation?

Without class syntax and private fields, the only way to hide state was to keep it as a local variable of a function and expose it only through inner functions (closures) returned in an object literal.

6. What is the type of `(a: string, b: number) => { return a.length > b; }`?

`(a: string, b: number) => boolean`.

7. Draw the frames for the moment `multiply` is executing in example 5.3. What is F2's P and RA?

F0 (globals), F1 square (P F0, RA 12, a 12), F2 multiply (P F0, RA 9, a 12, b 12). P is F0 because `multiply` was defined at global scope; RA is 9 because the call is on line 9.

L05.5 6. HTTP and the Rise of the Planet of the Apps

6.1 HTML vs HTTP

- **HTML** specifies the syntax and semantics of web page **content**. Associated CSS controls rendering; associated TypeScript/JavaScript provides front-end programmability (event handling).
- **HTTP** is a **request/response protocol**. A protocol is a formal set of rules for communicating over a network that specifies **message syntax and semantics**.
- HTTP can be used for **any kind of resource**, not just HTML. A resource is named by the **URL path**.

6.2 A basic HTTP exchange

1. **Establish a connection.** This is TCP's job. It needs the **hostname and port number**. Default port is **80** for non-secure HTTP and **443** for HTTPS.
2. **Client sends a request.** (Who is the client? The browser, or `curl`, or any program making the request.)
3. **Server sends a response.**
4. Possibly repeat steps 2 and 3.
5. **Close the connection.**

Typical use pattern: a first exchange retrieves the HTML page; subsequent exchanges retrieve associated content: stylesheets, JavaScript, images.

6.3 HTTP is stateless

STATELESS. Each request/response is **independent**. There is no notion of a persistent conversation between client and server: the server does not remember your previous request.

This was the attendance-check discussion, so be ready to argue both sides:

- **Why it is good:** simple to implement; servers do not have to store per-client state, so they scale easily and any server in a pool can answer any request; a failed request can just be retried; caching is easy.
- **Why it is bad:** anything that needs continuity (login sessions, shopping carts, multi-step forms) has to be re-sent with every request or reconstructed by some added mechanism. That mechanism is cookies (6.8).

6.4 HTTP request format

Three parts:

1. **Request-line**
2. **Header section**: one header per line; an **empty line indicates the end of headers**.

3. **Message body** (a.k.a. payload): may or may not exist. Often absent for simple retrieval (GET) requests.

```
GET /path/to/resource HTTP/1.1
Host: server
User-Agent: Mozilla/5.0 ...
Accept: text/html
Cookie: session=abc123
                                  <- empty line ends the headers
(optional body, e.g. form data for POST)
```

Request line: `METHOD RESOURCE VERSION`

- **METHOD**: one of a well-defined set of operations: **GET, POST, PUT, DELETE, OPTIONS, HEAD, TRACE, CONNECT**. Generally either GET or POST. **HEAD** is sometimes used to get information about a resource without actually getting the resource.
- **RESOURCE**: the **path portion of the URL**. The entire URL is also legal.
- **VERSION**: the string literal `HTTP/1.1`.

Worked example from the slide: for `http://server:port/path/to/resource`, TCP opens a connection to **server** at **port** (80 if none given), and the request line is `GET /path/to/resource HTTP/1.1`.

Headers

- Types: **general** headers, **message** headers (specific to requests or replies), **entity** headers (information about the message body).
- Syntax: `HEADER: VALUE`, one per line. Examples: `Date: Wed, 28 Sep 2011 5:30:00 GMT`, `Content-type: text/html`. Reference: MDN HTTP headers.

Common request header	Purpose
`User-Agent`	Describes the client making the request (browser name/version, OS).
`Host`	**Required in HTTP/1.1.** Names the server the client intended to contact. Facilitates **multi-hosting**: one web server handling requests for several different entities/domains needs the Host header to know which one is intended.
`Accept`	Indicates acceptable formats for the reply (e.g. `text/html`, `application/json`).
`Cookie`	Cookie data associated with the requested resource (sent back to the server that set it).

6.5 HTTP reply format

Also three parts: **status-line**, **header section**, **message body**.

```
HTTP/1.1 200 OK
Date: Wed, 28 Sep 2011 5:30:00 GMT
Content-type: text/html
Content-length: 1234

<!DOCTYPE html> ...
```

Status line: `VERSION CODE REASON`

- **VERSION**: `HTTP/1.1`.
- **CODE**: a 3-digit number whose **first digit indicates the general response type**.
- **REASON**: a human-readable phrase. **Not officially standardized** (can be anything), but some phrases are ubiquitous, like "File Not Found" / "Not Found" for 404.

Class	Meaning	Well-known examples (beyond the slide, but worth knowing)
1xx	Informational	100 Continue, 101 Switching Protocols
2xx	Success	200 OK, 201 Created, 204 No Content
3xx	Redirection	301 Moved Permanently, 302 Found, 304 Not Modified
4xx	Client error	400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found
5xx	Server error	500 Internal Server Error, 502 Bad Gateway, 503 Service Unavailable

6.6 The evolution of server-side programming: forms and CGI

"In the beginning, there were forms."

- Form **input elements** create a user interface.
- Input values are encoded **as part of the URL** (for GET, e.g. `/search?q=cats`) or **as the request payload** (for POST).
- **Common Gateway Interface (CGI)** invokes server-side processing: a URL path is mapped to an **external process** that runs, reads the inputs, and writes back an HTML page.

6.7 Cookies and server-side state

- **Intra-session** state (within one visit) is possible with stateless HTTP, but it requires being **passed back and forth** in every request, and it is not suitable for information that should not be exposed to the client.
- **Inter-session** state (remembering you next week) is **not possible** with stateless HTTP alone.
- **Cookies** provided a mechanism to add state: the server sets a small value that the client stores and sends back with each later request (the `Cookie` header). Two uses:
  - As an **index to state stored on the backend**, e.g. a shopping cart id.
  - As a **certificate of authentication** (proof you logged in). This requires **encrypted communication** (HTTPS) to prevent **man-in-the-middle** attacks that steal the cookie.

6.8 Enter AJAX

- Early web applications were clunky: they **reloaded the DOM on every interaction**, with noticeable round-trip latency to the server.
- **AJAX (Asynchronous JavaScript and XML)** was introduced by Microsoft around 1999/2000 via the **XMLHttpRequest (XHR)** mechanism: construct and initiate HTTP requests from JavaScript, with the **response handled asynchronously in JavaScript**. This enabled communication with the backend **without a full DOM reload**.
- Early famous use case: **search term autocompletion by Google**.

6.9 Rise of the Planet of the Apps

THE TRANSFORMATION. The Web changed from a **document-based request service** into a **distributed application framework**:

- The web application's HTML skeleton loads **once** at the beginning.
- All subsequent communication retrieves and updates **application state** from the backend **asynchronously**.
- The backend becomes a **web service**: URLs are understood as **endpoints**, and replies are **NOT new HTML pages but simply data**.

Current state of the art: web servers primarily serve as **endpoint API implementations first** and as static document servers secondarily. XHR was **callback-based** (you created an object representing the request, which encapsulated callbacks invoked when the response arrived). The **modern approach uses Promises**: `fetch()` and `async`/`await`, which is exactly L06.

6.10 Check yourself

1. Write the request line and the minimum required header for fetching `http://example.com/docs/index.html`.

`GET /docs/index.html HTTP/1.1` followed by `Host: example.com` (required in HTTP/1.1), then an empty line. TCP connects to port 80 since none was given.

2. What are the three parts of an HTTP request and of an HTTP reply?

Request: request-line, header section, message body (optional). Reply: status-line, header section, message body.

3. What does "HTTP is stateless" mean, and how do cookies compensate?

Each request/response is independent; the server keeps no persistent conversation. Cookies let the server hand the client a token that is sent back with every request, serving as an index to backend state (cart id) or an authentication certificate.

4. Why is the Host header required in HTTP/1.1?

To support multi-hosting: one server may serve several domains, so it needs to know which one the client intended.

5. Classify: 201, 302, 404, 503, 100.

201 success (2xx), 302 redirection (3xx), 404 client error (4xx), 503 server error (5xx), 100 informational (1xx).

6. Is the reason phrase in a status line standardized?

No. It is human-readable and can be anything, though some phrases like "Not Found" for 404 are ubiquitous. Programs should rely on the numeric code.

7. What did AJAX/XHR change about how web pages interact with servers?

JavaScript could construct HTTP requests and handle responses asynchronously, updating the page without a full DOM reload, which turned pages into applications talking to backend endpoints that return data rather than HTML.

8. Default ports for HTTP and HTTPS?

80 and 443.

9. What is HEAD for?

Getting information (headers) about a resource without retrieving the resource body.

L06 7. Asynchronous TypeScript

7.1 API endpoints and JSON

Resources are not limited to files: we can request **data** too. We request data from a server by making a request to a certain route, known as an **API endpoint**.

> API (Application Programming Interface). The entry point for communication between different software systems.

You can hit an endpoint from the browser's address bar or with the system utility `curl`:

```
curl https://comp426-apis.vercel.app/api/cs-organizations
```

Data often comes back as **JSON (JavaScript Object Notation)**, which specifies data in a **key-value pair** format. JSON makes it easy to retrieve data from servers and represent it as objects in JS/TS code, but it is **language agnostic**: apps in other languages can load JSON too.

```
[
  { "name": "Carolina Women in Computing", "abbreviation": "CWiC" },
  { "name": "App Team Carolina", "abbreviation": "ATC" }
]
```

7.2 Why we cannot just call fetch and use the result

It is tempting to write `const data = fetch(url).json();`. That does not work, for two reasons that build on each other.

> JavaScript / TypeScript is single-threaded. There is only **one call stack**, so only one operation can be performed at a time. Fetching data from an API requires connecting to the internet, requesting data, and retrieving it, which can take a long time depending on connection speed and response size.

```
console.log("Start");
const data = fetch(url).json();   // imagine this blocked for 3 seconds...
console.log(data);
console.log("End");
```

If this call were blocking, the program could do **nothing else** while it ran. Not even moving the mouse: **the entire page freezes**. That is obviously very bad, so we need a way to prevent blocking operations from blocking the single thread: **asynchronous programming**.

ASYNCHRONOUS EVENT MODEL. (1) Start an expensive process and leave it to the browser to work on in the background. (2) Continue executing code. (3) When the expensive process is finished, perform an action with its result.

7.3 Pattern 1: callbacks

Callbacks are functions provided to a blocking (i.e., asynchronous) operation, to be called after the operation is done.

- Event handlers for UI elements are a sort of callback: registered with `addEventListener`, called in the future when the event occurs.
- The original XHR mechanism took this approach: create an XHR object; configure it with the HTTP method and URL; set request headers; **attach callbacks** to handle the result; initiate the request, which **returns immediately**; when the request completes in the future, the callback is executed.

```
const xhr = new XMLHttpRequest();
xhr.open("GET", "https://comp426-apis.vercel.app/api/cs-organizations");
xhr.onload = () => {              // callback: runs later, when the response arrives
  const orgs = JSON.parse(xhr.responseText);
  console.log(orgs);
};
xhr.send();                       // returns immediately
console.log("request sent");      // prints BEFORE the orgs
```

7.4 Pattern 2: promises

An alternative to callbacks. Asynchronous (blocking) operations **return a Promise object**. Promise objects have methods for specifying what to do after the promise is **fulfilled** (`.then`) or for handling errors (`.catch`). Promises can be **chained** if a `then` handler returns another promise. This is what `fetch()` does.

```
fetch("http://some/api/endpoint")
  .then((result) => {
    // handle the case when a value is successfully produced
    return fetch("http://some/other/endpoint");   // returning a promise chains it
  })
  .then((result) => {
    // handle the second result
  })
  .catch((error) => {
    // handle the case where an error occurs anywhere along the way
    console.error(error);
  });
```

- A promise is an object representing a value that will exist *later*. `Promise<T>` will eventually produce a `T`.
- `.then` takes a function of type `(result: T) => void` (or one returning another value/promise). **Once the asynchronous process completes, the Promise calls this function and passes in the result.**
- The passed-in function is **not added to the call stack until after the async process completes**. Everything after the `.then(...)` line runs first.
- Promise states (standard terminology): *pending* → *fulfilled* (with a value) or *rejected* (with an error). Printing a promise before it settles shows `Promise { <pending> }`.

7.5 Pattern 3: async / await

Makes writing and reading asynchronous code much, much easier.

- `await` can be used with **any function that returns a Promise**.
- `await` effectively **blocks your code from continuing until the Promise is resolved** (only the code inside that async function is paused; the rest of the page keeps running).
- The result of `await` is the **resolved value** from the Promise.
- Because `await` is blocking, any function that uses `await` must be marked **`async`**.
- Marking a function `async` **converts it into one that returns a Promise**. A `return "x"` inside an async function becomes a `Promise<string>` that resolves to "x".

```
const load = async (): Promise<string[]> => {
  const result = await fetch("https://comp426-apis.vercel.app/api/cs-organizations");
  const json = await result.json();     // .json() is ALSO async and must be awaited
  return json.map((o: { name: string }) => o.name);
};
```

WATCH OUT. `fetch` returns a promise of a *Response*. Getting the body as JSON needs a *second* asynchronous call, `.json()`, which also returns a promise. Two awaits (or two `.then`s), not one. `const data = fetch(url).json()` is wrong on both counts.

7.6 Tracing async code: the core example

```
console.log("Start");

const performExpOp = async () => {
  await expensiveOperation();
  return "I love COMP 426!";
}

const result = performExpOp();   // kick off the expensive operation
console.log("End");              // continue running while we wait
console.log(result);
```

Output:

```
Start
End
Promise { <pending> }
```

Not "I love COMP 426!". Execution sequence: `console.log("Start")` → `performExpOp()` is called and the async process is kicked off; the call returns a pending promise *immediately* → `console.log("End")` → `console.log(result)` prints the pending promise. Some time later the async process completes, but by then the script has already moved on. The result's intended value only exists after the process completes, so we can only do something with it **after** it is found: via `.then` or `await`.

```
console.log("Start");
performExpOp()
  .then((result) => {
    console.log(result);        // runs only after the async process completes
  });
console.log("End");
```

```
Start
End
I love COMP 426!
```

7.7 The rules for predicting output order

1. **Synchronous code runs top to bottom first, all of it.** Calling an async function runs its body synchronously *up to the first* `await`, then returns a pending promise and control comes back to the caller.
2. Any `.then` callback (or code after an `await`) is deferred until its promise resolves. It cannot run until the current synchronous code has completely finished, *even if the promise resolves instantly.*
3. Deferred callbacks then run in the order their promises **resolve** (by completion time), not the order they were started.

4. When a `.then` callback itself starts new async work, the same rules apply recursively: the callback's synchronous lines run immediately, and its nested `.then` waits.
5. Timing durations add up along a chain: an await of 100ms followed by a call that takes 300ms completes at 400ms.

7.8 Practice: cat and dog

```
const cat = async () => {
  await /* action takes 500ms */;
  return ...;
}
const dog = async () => {
  await /* action takes 100ms */;
  return ...;
}

console.log("One");
cat().then(result => { console.log("Cat") });
console.log("Two");
dog().then(result => { console.log("Dog") });
console.log("Three");
```

```
One
Two
Three
Dog
Cat
```

"One", "Two", "Three" are synchronous and print immediately; the two calls only *start* their timers. Dog finishes at 100ms and Cat at 500ms, so "Dog" prints before "Cat" even though cat was called first.

7.9 Attendance question: foo and bar (nested async functions)

```
const foo = async () => {
  await /* action runs for 100ms */;
  return async () => {
    await /* action runs for 300ms and returns */;
  }
}
const bar = async () => {
  await /* action runs for 200ms and returns */;
}

bar().then(_ => { console.log("bar returned!"); });
foo().then(res => {
  res().then(_ => {
    console.log("BOO!");
  });
  console.log("foo returned!");
});
```

```
foo returned!
bar returned!
BOO!
```

Timeline, with t = 0 when the script starts: both `bar()` and `foo()` are kicked off right away (bar will finish at 200ms, foo at 100ms). Nothing prints synchronously. At **t = 100ms** foo's promise resolves with `res`, a function. Its `.then` runs: it calls `res()`, which starts a 300ms action (finishing at t = 400ms), and then *immediately* logs "foo returned!" because the nested `.then` is deferred. At **t = 200ms** bar resolves: "bar returned!". At **t = 400ms** `res()` resolves: "BOO!". The slide note reminds you that `foo()` returns a `Promise<() => Promise<void>>`: a promise of a function that itself returns a promise.

EXAM TIP. For any "what is printed" question: (1) write down every synchronous line in order; (2) list each async operation with its start time and duration; (3) sort completions by finish time; (4) for each completion, run its callback's synchronous lines and add any new async work to the list. A short timeline table is the safest way to avoid mistakes.

7.10 Putting it together: fetch with async/await

`fetch` is an asynchronous function that returns a promise. On the result you must also call `.json()`, which is async too:

```
const result = await fetch(...);
const json = await result.json();
```

The in-class "Your turn" built a page listing CS organizations. A complete solution, using L04 DOM methods:

```
type Org = { name: string };

const showOrgs = async () => {
  const response = await fetch("https://comp426-apis.vercel.app/api/cs-organizations");
  const orgs: Org[] = await response.json();
  const container = document.getElementById("orgs")!;
  for (const org of orgs) {
    const div = document.createElement("div");
    const p = document.createElement("p");
    p.innerText = org.name;
    div.append(p);
    container.append(div);
  }
};

showOrgs();   // or attach to a button's click listener
```

The equivalent with promise chaining:

```
fetch(url)
  .then((response) => response.json())     // returns a promise -> chained
  .then((orgs: Org[]) => { /* build the DOM */ })
  .catch((err) => console.error(err));
```

Error handling with async/await uses ordinary `try { ... } catch (err) { ... }` around the awaits. `await` outside an async function is a compile error in this course's setup (top-level await exists only in ES modules), so wrap it in an async function.

7.11 Check yourself

1. Why can't JavaScript just wait for a network request to finish?

> JS is single-threaded with one call stack; a blocking wait would freeze the entire page, including UI interaction, until the response arrives.

2. What does marking a function `async` change about its return type?

> It always returns a Promise; a returned value `T` becomes `Promise<T>`.

3. What does `await` do, and where may it be used?

Pauses that async function until the promise resolves and yields the resolved value; usable on any promise-returning function, only inside an `async` function.

4. What prints?

```
const f = async () => { console.log("A"); await delay(50); console.log("B"); };
console.log("C");
f().then(() => console.log("D"));
console.log("E");
```

C, A, E, B, D. The body of `f` runs synchronously up to the first await (printing A), then E prints, then after 50ms B prints and the promise resolves, so D prints.

5. What prints?

```
const slow = async () => { await delay(300); return 1; };
const fast = async () => { await delay(100); return 2; };
slow().then(v => console.log("slow", v));
fast().then(v => { console.log("fast", v); slow().then(v2 => console.log("again", v2)); });
console.log("done");
```

done; fast 2 (t=100); slow 1 (t=300); again 1 (t=400, since the second slow call starts at 100ms).

6. Two things wrong with `const data = fetch(url).json();`?

`fetch` returns a promise, not a Response, so `.json()` cannot be called on it directly; and `.json()` itself returns a promise, so the result must also be awaited or chained with `.then`.

7. Compare callbacks and promises as approaches to async code. Which did XHR use, and which does fetch use?

Callbacks: pass a function to be called later (event handlers, XHR). Promises: the operation returns an object with `.then`/`.catch` methods that can be chained; `async`/`await` is syntax on top of promises. XHR was callback-based; fetch returns promises.

8. What is JSON, and why does it matter that it is language agnostic?

JavaScript Object Notation: a key-value text format for data. Any language can parse it, so a server written in Python can talk to a TS frontend.

R02 · R03 8. TypeScript syntax review (from the readings)

The lectures assume the readings. Anything below could show up as a "what does this print," "find the compile error," or "write a function" question, and the memory-diagram questions depend on being fluent with arrow functions and object literals.

8.1 Types, variables, constants

Java	TypeScript	Notes
`int`, `double`	`number`	One numeric type for integers and decimals (64-bit floating point).
`boolean`	`boolean`	`true` / `false`.
`String`	`string`	Lowercase built-in type.
`int x = 88;`	`let x: number = 88;`	`let` keyword; annotation *after* the name.
`final int x = 88;`	`const x: number = 88;`	`const` replaces `let`; cannot be reassigned.
—	`let x = 88;`	Type **inferred** as number when annotation omitted. Annotations are still strongly encouraged.

`const` prevents reassignment of the variable, not mutation of the object or array it points to: `const a = [1]; a.push(2);` is fine, `a = [3]` is not.

8.2 Arrays

TS arrays behave like Java `List` (growable) but are indexed with `[]` like Python lists.

```
let dogs: string[] = ["Corgi", "Lab"];
dogs.push("Husky");                     // add to end
dogs[2] = "Samoyed";                    // replace by index
dogs.splice(dogs.indexOf("Samoyed"), 1); // remove 1 element starting at that index
dogs.splice(1, 1);                      // remove by index
let corgi: string = dogs[0];            // access
dogs.pop();                             // remove and return last item
dogs.length;                            // length is a field, not a method
```

8.3 Conditionals and loops

- `&&`, `||`, `!` as in Java, with short-circuiting. `if`/`else` and `while` are identical to Java.
- Counting loop: `for (let i = 0; i < 10; i++) { ... }` (note `let`).
- Collection loop: `for (let name of names) { ... }`. TS uses `of` where Java uses `:`. (`for ... in` iterates keys/indices, a common bug: use `of` for values.)

- Ternary: `condition ? exprIfTrue : exprIfFalse`, an expression, e.g. `let hour = isWeekday ? 10 : 12;`.

8.4 Functions and arrow functions

```
// Traditional function
function greet(name: string): string {
  return "Welcome, " + name + "!";
}
// void return type is optional
function doSomething(): void { }
```

```
// Arrow function: a function VALUE stored in a
variable
let greet = (name: string): string => {
  return "Welcome, " + name + "!";
};
greet("Jade");   // called the same way
// Concise body: implicit return, no braces
const double = (n: number) => n * 2;
```

- Methods are called on an object (`object.method()`); functions are called standalone.
- Two caveats from R02: arrow functions bind `this` at definition time (so do not use them as class methods that rely on `this`), and arrow functions **cannot be used as constructors** (`new` on one throws a TypeError).
- **Function literal** (R03): the anonymous arrow expression itself, `(num: number): number => { return num * 2; }`, is a value. Unused, it just disappears; assigned to a variable, it can be called. An *immediately invoked* literal looks like `((n: number): void => { console.log(n); })(423)`.
- **Function types**: `(param: type, ...) => returnType`. So `doubleNumber` has type `(num: number) => number`. Type aliases help: `type NumberTransformer = (num: number) => number;`.

8.5 Classes, interfaces, structural typing

```
class Student {
  public name: string;          // fields: no let
  public year: number;
  private address: string;

  constructor(name: string, year: number, adr: string) {   // 'constructor' keyword
    this.name = name;
    this.year = year;
    this.address = adr;
    this.welcome();
  }

  public welcome() {            // methods: no 'function' keyword
    console.log("Hello, " + this.name + "!");
  }

  public static yearToString(year: number): string {
    return year == 1 ? "Freshman" : year == 2 ? "Sophomore" : year == 3 ? "Junior" : year ==
4 ? "Senior" : "Oops...";
  }
}
let noah: Student = new Student("Noah", 3, "Columbia St");

interface Person { name: string; }
class Student2 implements Person { name = ""; }
```

STRUCTURAL TYPING ("DUCK TYPING"). TypeScript treats objects as the same type if they have the **same structure**, not merely the same name. Java is **nominally typed** (same name or inheritance required). So an object literal `let person: Person = { name: "Charles" };` is a valid `Person` without any class implementing the interface. "If it's a goose that looks like a duck, it's a duck."

8.6 Enums, type aliases, unions, intersections, generics

```
enum Direction { Up, Down, Left, Right }         // options capitalized like words, not ALL
CAPS
const isVertical = (d: Direction) => d == Direction.Up || d == Direction.Down;

type Rating = number;                                // alias for a primitive
type Student = { name: string; year: number; };     // alias for an object shape
let s: Student = { name: "Noah", year: 3 };

let id: string | number;                             // union: either type
id = "abc"; id = 4;   // ok        id = true;       // error
const getUser = (n: number): string | null => n === 1 ? "Alice" : null;
type Day = "Monday" | "Tuesday" | "Wednesday";      // string-literal union instead of an
enum

type A = { a: string }; type B = { b: number };
type AB = A & B;                                     // intersection: must have both
let ab: AB = { a: "hi", b: 42 };

class LinkedList<T> {                                // generic type parameter
  private value: T;
  private next: LinkedList<T> | null = null;
  constructor(value: T) { this.value = value; }
  getValue(): T { return this.value; }
}
let names: LinkedList<string> = new LinkedList("a");
```

8.7 Spread, rest, destructuring

```ts
// Spread (...) expands an array or object into individual parts
const nums = [1, 2, 3];
sum(...nums);                          // same as sum(1, 2, 3)
const both = [...nums, 4, 5];          // [1, 2, 3, 4, 5]
const merged = { ...{ a: 1 }, ...{ b: 2 } };   // { a: 1, b: 2 }

// Rest (...) in a parameter list collects any number of args into an array
function sumAll(...numbers: number[]): number {
  let total = 0;
  for (let n of numbers) total += n;
  return total;
}
sumAll(1, 2);          // 3
sumAll(...nums);       // 6

// Destructuring pulls values out into variables
const [first, second] = nums;                       // 1, 2
const person = { name: "Alice", age: 30 };
const { name, age } = person;                       // order does not matter for objects
const { name: personName } = person;                // rename
```

8.8 Odds and ends

- Comments: `//` and `/* */` as in Java. Printing: `console.log(value)`; in a browser it appears in the dev tools console.
- Object literals `{ key: value }` create objects directly (L05 uses them to build "objects" from closures).
- `===` and `!==` compare without type coercion and are what you should use; `==` coerces (`"2" == 2` is true). The readings use `==` in examples, but strict equality is standard practice.
- `null` vs `undefined`: a missing property reads as `undefined` (L03's `user.address` example in JS); `null` is an explicit "no value," often in union return types like `string | null`, and `getElementById` returns `null` when nothing matches.

8.9 Check yourself

1. Rewrite `function square(n: number): number { return n * n; }` as an arrow function and give its type.

`const square = (n: number): number => n * n;` Type: `(n: number) => number`.

2. Find the compile error: `let count: number = "3";`

A string is assigned to a variable annotated as number. Static typing rejects it at compile time.

3. Why is `let p: Person = { name: "Charles" }` legal in TS but the equivalent not in Java?

TS is structurally typed: any object with the right shape counts as a Person. Java is nominally typed and needs a class that implements the interface.

4. What is the difference between spread and rest, given both use `...`?

Spread expands an array/object into separate items at a call site or in a literal. Rest, used in a parameter list, collects any number of arguments into an array.

5. Write a type alias for a function that takes a string and returns a boolean, and a function of that type.

```
type Pred = (s: string) => boolean; const isLong: Pred = (s) => s.length > 5;
```

6. Output of `const { b, a } = { a: 1, b: 2 }; console.log(a, b);`?

`1 2`. Object destructuring matches by property name, not position.

7. Give a `mapNumbers([1,2,3], makeMultiplier(4))` result using R03's functions.

`[4, 8, 12]`.

9. Practice exam (mixed, all topics)

Aim for 60 minutes without notes, then check. Question styles mirror the in-class activities: definitions, short explanations, tracing, drawing, and short code writing.

Part A: short answer and definitions

A1. Describe, in order, every step and every message from typing `https://unc.edu/about` in a browser to seeing the page rendered, including what the DNS server, the web server, and the browser each do. [L01, L02, L03]

> Browser parses the URL (protocol https, domain unc.edu, path /about). It asks a DNS server for unc.edu's IP; DNS replies with the IP. The browser opens a TCP connection to that IP on port 443 and sends an HTTPS-encrypted HTTP request `GET /about HTTP/1.1` with `Host: unc.edu`. The server responds with the HTML. The browser parses the HTML, builds the DOM, and on reaching `<link>`/`<script>` tags in the head requests the CSS and JS files (further request/response exchanges). It renders the page according to the CSS and runs each JS file as encountered.

A2. Give the slide definition of a protocol, then name the two protocols at the heart of the internet and the three-word description of the service they provide. [L01]

> A set of rules that define exactly how a communication service is implemented. TCP and IP. Process-to-process, full duplex, byte stream.

A3. What are the four layers of the CSS box model, from outside in, and which two CSS properties size only the innermost layer? [L02]

> Margin, border, padding, content. `width` and `height` size only the content.

A4. Explain "statically typed" and give one concrete bug TS catches at compile time that JS lets through at runtime. [L03]

> Types are declared or inferred at compile time so the compiler can type-check before the code runs. Example: `add("2", "3")` for a function declared with number parameters (JS yields "23"), or reading `user.address` on an object without that property (JS yields undefined).

A5. Define: transpilation, build tool, runtime environment, package manager. Name the course's instance of each. [L03]

> Transpilation: compiling one high-level language into another (TS → JS) so the browser can run it. Build tool: software that performs the build (transpile, bundle, prepare for deployment): Vite. Runtime environment: a space where code runs, here one that runs JS outside the browser: Node.js. Package manager: installs packages into an environment: npm (cf. pip for Python).

A6. List the five rules for drawing a DOM tree. [L04]

Starts with a Document node; the root `<html>` is labelled documentElement; child elements are children in order; text is its own node, child of its element; attributes are associated with the element but are not children.

A7. What is a closure, and what in the memory-diagram notation makes closures work? [L05]

A function that retains access to variables from the scope in which it was defined even after that scope's function has returned. In the notation, each heap function records P, the frame it was defined in; every call frame for that function copies that P, giving it access to those variables.

A8. State the request line format and the status line format of HTTP/1.1, and give one example of each. [L05.5]

Request line: `METHOD RESOURCE VERSION`, e.g. `GET /index.html HTTP/1.1`. Status line: `VERSION CODE REASON`, e.g. `HTTP/1.1 404 Not Found`.

A9. Why is JavaScript single-threadedness the motivation for asynchronous programming? [L06]

With one call stack, a long operation like a network fetch would block everything, freezing the page. Async lets the browser handle the slow work in the background while code keeps running, then invokes a callback/then/await continuation when the result is ready.

A10. Three differences between XHR-style callbacks and promises. [L05.5, L06]

Callbacks are attached to a request object before sending; promises are returned by the operation. Promises can be chained with `.then` when a handler returns another promise; callbacks nest. Promises centralize error handling with `.catch`; and `async`/`await` syntax works on promises, not raw callbacks.

Part B: HTML and CSS

B1. For the HTML below, which paragraphs are red? Which are bold? What is the font style of the last one?

```
<div class="card important">
  <p class="text">One</p>
  <p>Two</p>
  <div><p class="text">Three</p></div>
</div>
<p class="text">Four</p>
```

```
.card p        { color: red; }
.card > p      { font-weight: bold; }
div.important p + p { font-style: italic; }
```

[L02]

Red: One, Two, Three (all descend from `.card`). Bold: One and Two only (direct children of `.card`; Three is inside an inner div; Four is outside). Italic: Two (a p immediately following a p inside div.important). Three is not italic: its immediately preceding sibling is nothing. Four is neither red, bold, nor italic.

B2. Draw the DOM tree for:

```
<body>
  <nav class="top"><a href="/">Home</a><a href="/about">About</a></nav>
  <main><h2>Title</h2><p>Some <strong>bold</strong> text</p></main>
</body>
```

[L04]

```
Document
└── documentElement <html>
    └── Element: <body>
        ├── Element: <nav>  ~~ Attribute: class, Value: "top"
        │   ├── Element: <a>  ~~ Attribute: href, Value: "/"
        │   │   └── Text: "Home"
        │   └── Element: <a>  ~~ Attribute: href, Value: "/about"
        │       └── Text: "About"
        └── Element: <main>
            ├── Element: <h2>
            │   └── Text: "Title"
            └── Element: <p>
                ├── Text: "Some "
                ├── Element: <strong>
                │   └── Text: "bold"
                └── Text: " text"
```

Note the paragraph has three children: text, strong, text.

B3. A div has `margin: 8px 16px; border: 1px solid; padding: 12px; width: 240px; height: 100px;`. Total width and height? [L02]

Two-value margin shorthand is vertical, horizontal: top/bottom 8, left/right 16. Width: 16+16+1+1+12+12+240 = 298px. Height: 8+8+1+1+12+12+100 = 142px.

B4. Write CSS to lay three `<section>` children of `#row` out side by side. [L02]

`#row { display: flex; flex-direction: row; }` (row is the default direction, so `display: flex;` alone suffices).

Part C: tracing and memory diagrams

C1. Draw the full memory diagram (stack and heap, with P, RA, RV) at the moment line 6 executes, and give the final value of `total`.

```
1  const base = 10;
2  const makeAdder = (n: number) => {
3    return (x: number) => x + n + base;
4  }
5  const add5 = makeAdder(5);
6  const total = add5(1);
```

[L05]

Heap: id:0 (P: F0, fn lines 2–4, param n). Frame F1 makeAdder (P F0, RA 5, n 5, RV id:1). Heap id:1 (P: F1, fn line 3, param x). F0: base 10, makeAdder id0, add5 id1. At line 6: frame F2 add5 (P: F1, RA 6, x 1). Inside, `n` is found in F1 (5) and `base` in F0 (10) by following P links. RV 16. `total` = 16.

C2. What does this print?

```
const make = () => {
  let items: string[] = [];
  return {
    add: (s: string) => { items.push(s); },
    size: () => items.length,
  };
};
const a = make();
const b = make();
a.add("x"); a.add("y"); b.add("z");
console.log(a.size(), b.size());
```

[L05]

`2 1`. Each `make()` call has its own frame with its own `items`; the returned object's functions close over that frame.

C3. What does this print?

```
const compose = (f: (n: number) => number, g: (n: number) => number) =>
  (x: number) => f(g(x));
const inc = (n: number) => n + 1;
const dbl = (n: number) => n * 2;
console.log(compose(inc, dbl)(5), compose(dbl, inc)(5));
```

[L05, R03]

11 12. `compose(inc, dbl)(5)` = inc(dbl(5)) = 11; `compose(dbl, inc)(5)` = dbl(inc(5)) = 12. `compose` is higher-order in both senses: takes functions, returns a function.

C4. Output order?

```
const a = async () => { await wait(300); console.log("a done"); return "A"; };
const b = async () => { await wait(100); console.log("b done"); return "B"; };
console.log("1");
a().then(v => console.log("then", v));
console.log("2");
b().then(v => { console.log("then", v); a().then(v2 => console.log("nested", v2)); });
console.log("3");
```

[L06]

1, 2, 3, b done (100ms), then B, a done (300ms), then A, a done (400ms: second a() started at 100ms), nested A.

C5. Output?

```
const f = async () => {
  console.log("f start");
  const r = await g();
  console.log("f got", r);
};
const g = async () => { console.log("g runs"); return 7; };
f();
console.log("after f");
```

[L06]

f start, g runs, after f, f got 7. `f` runs synchronously until its first await; evaluating `g()` runs g's body synchronously (no await inside), returning an already-resolved promise; the `await` still yields control, so "after f" prints before the continuation.

C6. What is printed, and why is it not the org list?

```
const orgs = fetch("https://comp426-apis.vercel.app/api/cs-organizations");
console.log(orgs);
```

[L06]

A pending Promise (`Promise { <pending> }`). `fetch` returns immediately with a promise of a Response; the data is only available after awaiting it and then awaiting `.json()`.

Part D: write code

D1. Write TS so that clicking any `<li>` inside `<ul id="menu">` turns that specific item's text red, using a single listener on the `ul`. [L04]

```
const menu = document.getElementById("menu")!;
menu.addEventListener("click", (event) => {
  const target = event.target as HTMLElement;   // the li that was clicked (bubbling)
  if (target.tagName === "LI") {
    target.style.color = "red";
  }
});
```

D2. Write an async function `loadNames(url)` that fetches JSON of the shape `[{ name: string }]` and returns a `string[]` of names. Then show how to call it and log the result. [L06]

```
const loadNames = async (url: string): Promise<string[]> => {
  const response = await fetch(url);
  const data: { name: string }[] = await response.json();
  return data.map((d) => d.name);
};
loadNames(API).then((names) => console.log(names));
// or, inside another async function: const names = await loadNames(API);
```

D3. Write `filterNumbers(nums, keep)` where `keep` is a function deciding whether to keep each number; give its full type annotations and call it to keep even numbers. [R03]

```
const filterNumbers = (nums: number[], keep: (n: number) => boolean): number[] => {
  const out: number[] = [];
  for (const n of nums) if (keep(n)) out.push(n);
  return out;
};
filterNumbers([1, 2, 3, 4], (n) => n % 2 === 0);   // [2, 4]
```

D4. Write a `counter()` factory returning an object with `increment()`, `decrement()`, and `value()` that share private state. [L05]

```
const counter = () => {
  let count = 0;
  return {
    increment: () => { count += 1; },
    decrement: () => { count -= 1; },
    value: () => count,
  };
};
```

D5. Write the raw HTTP request a browser would send for `http://api.example.com:8080/orgs?year=2026`, with the required header, and a plausible success status line for the reply. [L05.5]

```
GET /orgs?year=2026 HTTP/1.1
Host: api.example.com
```

TCP connects to api.example.com on port 8080. Reply status line: `HTTP/1.1 200 OK`.

10. One-screen cram sheet

Read this the night before. Every item is a term or fact stated on a slide.

L01 Internet. Network = 2+ computers connected. Internet = interconnected network worldwide. WWW = content-sharing system over the Internet. IP address identifies a device. Protocol = rules defining a communication service; layering = complex on simpler. TCP/IP = process-to-process, full duplex, byte stream. Client requests; server provides resources. HTTP = request + response. HTTPS = encrypted. URL = protocol :// domain / path. DNS = domain → IP ("address book").

L02 HTML/CSS. HTML = content + structure. Tags `<x>…</x>`; content between; nesting. `<html>` holds `<head>` (metadata) and `<body>` (content). Attributes; `class` groups, `id` unique. CSS = styling; rule = selector { property: value }. `#id`, `.class`, `type`, `E F` descendant, `E > F` child, `E + F` next sibling, `:hover`; compound read right to left. Layout viewport in px; origin top-left. Box model: margin, border, padding, content; width/height size content only. `display: flex` + `flex-direction: row|column`. Layout = HTML + CSS together.

L03 JS/TS/Build. JS adds functionality; `<script src>`. Load: HTML → DOM → request CSS/JS at head → render → run JS as encountered. TS JS; static (compile-time) vs dynamic (runtime) typing; catches ~20% of errors. Browser runs only JS (interpreted) → transpile TS→JS at build. gcc/a.out, Maven/JVM, Vite/browser. Vite = fast build tool + dev server. Node.js = JS runtime outside browser; environment holds packages; npm installs (like pip). `npm run dev` → localhost (127.0.0.1) : port.

L04 DOM/Events. DOM = tree data representation of page; elements as objects. Tree: Document → documentElement → elements; text nodes are children; attributes attached, not children. `document` global. `getElementById` (null if none), `getElementsByTagName`, `querySelector` (first), `querySelectorAll` (all; scoped to a node). Node vs element navigation (`childNodes`/`children`, `firstChild`/`firstElementChild`, …). `el.addEventListener("click", () => {})` = higher-order fn. `innerHTML` vs `innerText`. Event object: type, target, currentTarget, bubbles, cancelable, isTrusted, timeStamp.

L05 Functional. Stack frames: name (F#), P = defining frame, RA = return line, RV, locals. Heap: functions (id, P, lines, params), objects, strings. Higher-order fn = takes and/or returns a function. Closure = inner fn keeps access to defining frame's variables (counter prints 1,2,3). Paradigm = way of organizing code; FP models everything with functions. Object literals `{}`; encapsulation via closures (pre-`class` JS); get/set syntax. Event handlers are FP in action.

L05.5 HTTP. HTTP = request/response protocol for any resource named by URL path. Exchange: TCP connect (host, port 80/443) → request → response → repeat → close. Stateless. Request = request-line + headers + blank line + optional body. `METHOD RESOURCE HTTP/1.1`; methods GET POST PUT DELETE OPTIONS HEAD TRACE CONNECT. Headers `Name: value`; Host required (multi-hosting), User-Agent, Accept, Cookie. Reply = `HTTP/1.1 CODE REASON`; 1xx info, 2xx success, 3xx redirect, 4xx client err, 5xx server err; reason not standardized. Forms + CGI → cookies (backend index, auth cert; needs HTTPS) → AJAX/XHR (~2000, no DOM reload) → web as app framework: skeleton once, endpoints return data → fetch/promises/async-await.

L06 Async. API endpoint = route returning data; API = entry point between systems. JSON = key-value, language agnostic. JS single-threaded, one call stack → blocking freezes page → async model: start, continue, act on result later. Callbacks (handlers, XHR). Promises: returned by async ops; `.then(fn)` runs after resolve, `.catch`, chain by returning promises. `async` fn returns a Promise; `await` pauses until resolved, yields value, only inside async. Sync code first; then callbacks by completion time. `await fetch()` then `await .json()`.

R02/R03 TypeScript. `let/const`, `name: type`, `number/boolean/string`. Arrays like Java List: push, splice, pop, length, `[]` index. `for (let x of arr)`. `function f(a: T): R` or `const f = (a: T): R => {}`; function type `(a: T) => R`; type alias. Classes: fields without let, `constructor`, methods without `function`, `static`, access modifiers; interfaces; structural typing. Enums, unions `|`, intersections `&`, generics `<T>`, ternary, spread/rest `...`, destructuring.

Numbers to remember. Ports 80 (HTTP) / 443 (HTTPS). localhost = 127.0.0.1; example dev port 3000 (A02 uses 4260). Apple example IP 17.253.144.10. Total width = margins + 2 · border + paddings + content. Status code classes 1–5. TS prevents ~20% of JS errors. Grades: attendance 10, readings 10, coding 30, midterms 20 (10 each), project 30. Late: 48h window, 15% penalty, 3 dropped.

Sources: L01–L06 and L05.5 slide decks (Ketan Mayer-Patel, with Ajay Gandecha as collaborator by reference), readings R00–R03, and the course syllabus. Items marked "beyond the slides" are standard background included for completeness.

Built September 14, 2026 from the L01–L06 lecture decks and readings R00–R03 of COMP 426 (Fall 2026, UNC). Personal study material.