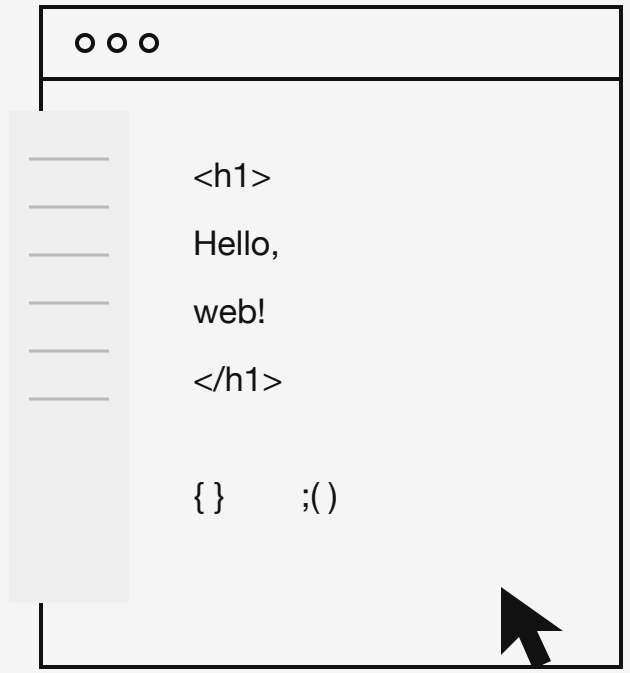


FIG. 01 — EVERY PAGE STARTS HERE

WEB DEVELOPMENT ACADEMY

Basic Web Application

September 12, 2026 | Lecture 01 · Online

What we cover today

01	**Web Foundations**		History · what a page is made of · DevTools · VS Code setup
02	**HTML5**		Syntax & tags · text, lists · attributes & ARIA · navigation · images · inputs
03	**CSS3**		Styling · selectors · box model · Flexbox · Grid · position & z-index · responsive
04	**JavaScript**		Console · variables · types · math · control flow · loops · arrays & objects · functions · DOM · Fetch
05	**Practice & Labs**		Four hands-on labs — from converting a Word article to live data from an API

01

Web Foundations

A short history, the moving parts of the platform, and the toolkit on your machine.

History · Anatomy of a page · Inspect · Libraries · VS Code

First the Internet — no web yet



1969 — ARPANET links its first four sites · 1983 — TCP/IP becomes the standard language

What people actually used: Email · FTP · Telnet · Newsgroups

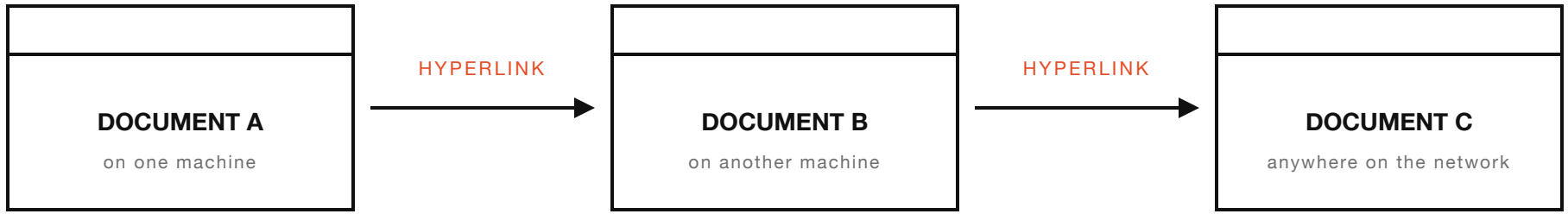
The key distinction: the Internet is the network; the Web is one service that runs on it — a sibling of Email or FTP, not the same thing.

The unsolved problem: documents live on separate machines, in different formats, with no way to point from one to another. Finding information means knowing exactly where it is.

SOURCES — INTERNET SOCIETY, A BRIEF HISTORY OF THE INTERNET · CERN WEB ARCHIVES

1989: linking documents at CERN

Thousands of researchers, and every document sits on a different system on a different machine. Tim Berners-Lee's proposal: documents that point to each other — a **web** of linked pages.



HTML
HyperText Markup Language — describes what a document contains: headings, paragraphs, links.

HTTP
HyperText Transfer Protocol — "give me this URL" → "here is the document".

URL
Uniform Resource Locator — one address that makes every document reachable.

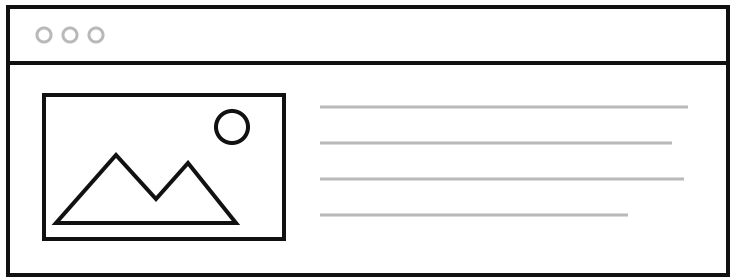
1990 — he builds the first browser (named WorldWideWeb) and the first server at CERN · **1991** — the first website goes live at `info.cern.ch`

Web = a web of linked documents. Notice what HTML was invented for: structure and meaning — the original web had no styling at all.

SOURCES — CERN WEB ARCHIVES · THE FIRST WEBSITE, INFO.CERN.CH

1993–1996: mainstream — then the browser wars

MOSAIC — 1993



The first browser people actually use — it shows **images next to text**, and the web stops being an academic tool. Much of its team goes on to found **Netscape (1994)**.

THE BROWSER WARS — NETSCAPE VS INTERNET EXPLORER

NETSCAPE NAVIGATOR
its own features

VS

INTERNET EXPLORER
its own, different features

```
document.all.hero    // exists only in Internet Explorer
document.layers.hero // exists only in Netscape
```

BEST VIEWED WITH INTERNET EXPLORER

1994 — the W3C is founded by Tim Berners-Lee: one home for the web standards (HTML, CSS), so the same document works in every browser.

Software-engineering lesson: when platforms implement different behavior, someone's users always get a broken product. Shared standards are what let one codebase serve everyone.

SOURCES — NCSA MOSAIC · W3C · NETSCAPE PRESS MATERIALS, 1995

Two problems give us CSS and JavaScript

PROBLEM 1 — THE WEB IS UGLY

HTML gets stretched past its job: `<font>` tags for color, tables drawn to fake page layout. Presentation is mixed into structure.

THE FIX — CSS, 1996

```
<p class="title">Hello</p>
.title { color: red; }
```

Structure stays in HTML, looks move to CSS — **Separation of Concerns**.

PROBLEM 2 — THE WEB IS STATIC

Click a button: nothing happens. A page is a poster that happens to be on a screen — no behavior of its own.

THE FIX — JAVASCRIPT, 1995

Brendan Eich builds the first version at Netscape in about **10 days**.

```
button.onclick = () => alert("Clicked!")
```

Pages start reacting to the user — documents become **applications**.

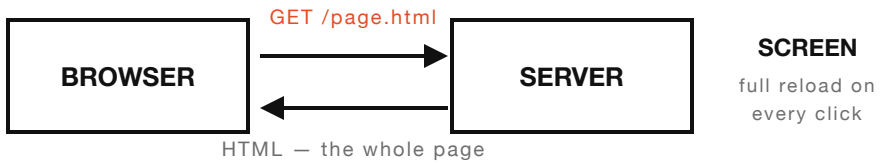
HTML	**CSS**	**JavaScript**
STRUCTURE	PRESENTATION	BEHAVIOR
the skeleton — content & meaning	the dressing — layout, color, spacing	the brain & muscle — logic & reaction

Notice the pattern: each language exists because a real problem forced it into being — that is still how the platform evolves today.

SOURCES — W3C CSS LEVEL 1, 1996 · NETSCAPE PRESS MATERIALS, 1995

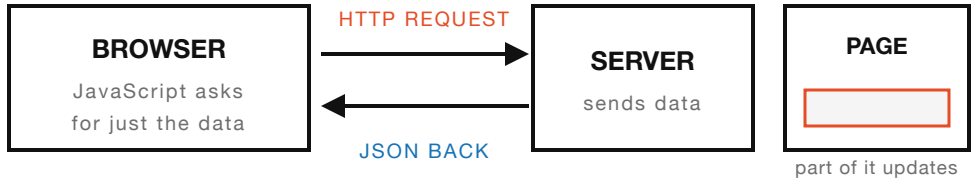
From read-only pages to the platform you use now

WEB 1.0 — READ (1990S)



SCREEN
full reload on every click

WEB 2.0 — READ + WRITE + AJAX (2000S)



Gmail (2004), Google Maps (2005) — users post, upload, comment.

2007
The iPhone puts a real browser in every pocket — responsive design and @media queries

2010s
HTML5 + browser APIs: <video>, <canvas>, camera, location, storage — a mini OS

TODAY
Frameworks: jQuery → React. You describe State → UI; it renders HTML, CSS, JS

Strip the layers away and the core is unchanged — URL → HTTP → HTML. React does not replace the three languages; it runs on them. The web grew from one question: "*How do we link documents between computers?*" Structure → HTML. Style → CSS. Behavior → JavaScript. That is this course.

SOURCES — W3C HISTORY · GOOGLE PRESS MATERIALS, 2004–2005 · MDN WEB DOCS

HTML, CSS & JavaScript — one body

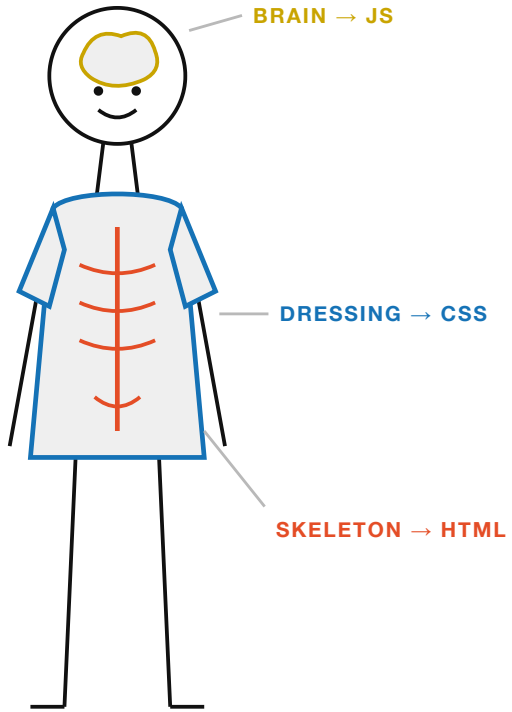


FIG. 02 — ONE BODY, THREE SYSTEMS

HTML

The skeleton

Structure that holds everything up — headings, paragraphs, images, buttons. Take it away and the body collapses.

CSS

The dressing

Clothes, colors and personality — how the body presents itself. Same skeleton, totally different style.

JAVASCRIPT

The brain & muscle

Thinks, decides, reacts and moves — behavior. It turns a body that just stands there into a living person.

Ask the class: a site with only HTML is a skeleton. Only CSS — clothes on the floor. Only JS — a brain in a jar. You need all three.

Under the hood: DevTools (Inspect)

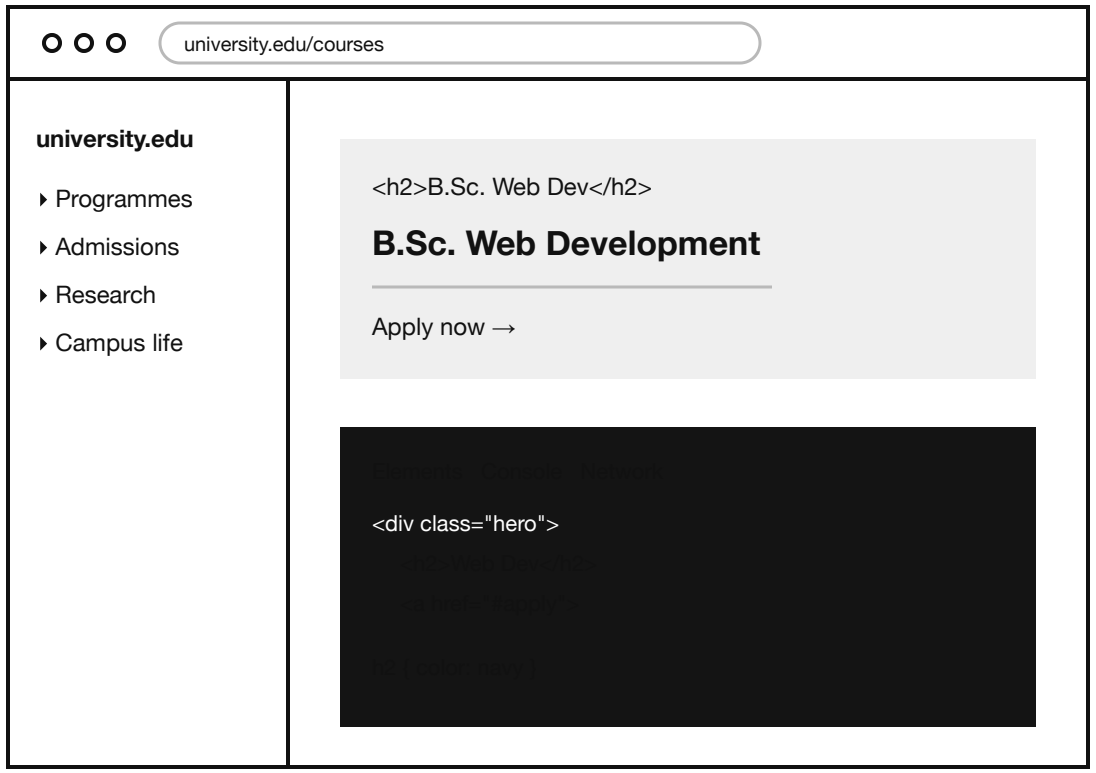


FIG. 03 — YOUR BROWSER, X-RAYED (LIVE DEMO IN CLASS)

Open it three ways

- **F12** — Windows / Linux
- **⌘⌥I** — macOS
- **Right-click → Inspect** — anywhere

Panels to know

- **Elements:** the live DOM — click any node, see the styles
- **Console:** run JavaScript right now, read errors
- **Network:** every file the page downloads, with timings
- **Device bar:** preview phone & tablet widths instantly

Golden rule: edits in DevTools are local and temporary — refresh and the real page returns. A perfect safe sandbox.

Prepare your environment

STEP 1

Install VS Code

Free at code.visualstudio.com. Built-in Emmet expands "!" into a full HTML5 boilerplate — type it, press Tab.

STEP 2

Add 3 extensions

Live Server (auto-reload browser) · Prettier (formats your code) · Auto Rename Tag (keeps tag pairs in sync).

STEP 3

Create a project folder

One folder per lab, with the three core files: index.html, styles.css, script.js.

STEP 4

Open with Live Server

Right-click index.html → "Open with Live Server". Save a file → the browser refreshes itself.

Check yourself: type an <h1>, save, and watch it appear without touching F5. If that loop works, your workflow is ready.

```
your first project

webdev-lab/
├── index.html    ← structure
├── styles.css    ← styling
└── script.js     ← behavior

Tip: open the FOLDER (not the file)
in VS Code — IntelliSense then
understands the whole project.
```



FIG. 04 — THE EDIT → SAVE → REFRESH LOOP

02

HTML5

The web's structure language: tags, attributes, and meaning.

Syntax | Text & lists | Attributes & ARIA | Links | Images | Inputs

Anatomy of an HTML page

index.html

```
<!DOCTYPE html>
<html lang="en">

  <head>
    <meta charset="UTF-8">
    <title>My first page</title>
  </head>

  <body>
    <h1>Hello, class!</h1>
    <p>My very own page.</p>
  </body>

</html>
```

In VS Code, type ! and press Tab — Emmet writes this whole boilerplate for you.



ELEMENT = OPENING TAG + CONTENT + CLOSING TAG

<!DOCTYPE html> — "modern HTML", not a 1998 quirk mode.

<html lang="en"> — root element; lang helps screen readers & translation.

<head> — metadata the visitor doesn't see. <meta charset="UTF-8"> enables every character.

<body> — everything visible: text, images, buttons.

Working with text

article.html

```
<h1>Main heading</h1>
<h2>Section heading</h2>

<p>Normal <strong>bold-important</strong>
and <em>emphasised</em> text.</p>

<p>Water is H<sub>2</sub>O,</p>
<p>and 2<sup>10</sup> = 1024.</p>

<p>Press <kbd>Ctrl</kbd> + <kbd>S</kbd>
to save.<br>New line,<br>same paragraph.</p>
```

RENDERED RESULT

Main heading

Section heading

Normal **bold-important** and *emphasised* text.

Water is H_2O, and $2^{10} = 1024$.

Ctrl + S to save.
New line,
same paragraph.

↑ *<hr> — a thematic break between topics*

h1–h6: six heading levels, h1 is the most important — use ONE per page. / carry meaning for screen readers; <b>/<i> are just looks.

Lists: ordered & unordered

lists.html

```
<ul>
  <li>Morning: HTML drills</li>
  <li>Afternoon: CSS layouts
    <ul>
      <li>Flexbox</li>
      <li>Grid</li>
    </ul>
  </li>
</ul>

<ol start="3">
  <li>Register the domain</li>
  <li>Deploy the site</li>
</ol>
```

- Morning: HTML drills
- Afternoon: CSS layouts
  - Flexbox
  - Grid

---

3. Register the domain
4. Deploy the site

`ul = dots · ol = numbers · nesting goes inside <li>`

Choosing: for menus, features, anything unordered · for steps, rankings, recipes · <ol start="3"> begins counting at 3.

Attributes & accessibility (ARIA)

```
< img  src= "cat.jpg"   alt= "A sleeping orange cat"   width="300" >
```

ELEMENT — <IMG> IS EMPTY, NO CLOSING TAG

ATTRIBUTE = NAME + VALUE IN QUOTES

ALT — READ ALOUD TO BLIND USERS, SHOWN WHEN THE IMAGE FAILS

SIZE IN PIXELS — PREVENTS LAYOUT JUMP

Global attributes — every element

- **id="header"** — unique; one per page, for #links & JS
- **class="card"** — reusable style hook, shared by many
- **title="…"** — tooltip on hover
- **lang, style, hidden** — language, inline CSS, visibility

ARIA — when HTML alone isn't enough

- **role="button"** — a behaving like a button (last resort)
- **aria-label** — name an icon-only link for screen readers
- **aria-hidden="true"** — hide decoration from assistive tech
- **Rule:** semantic HTML first — ARIA is seasoning, not flour

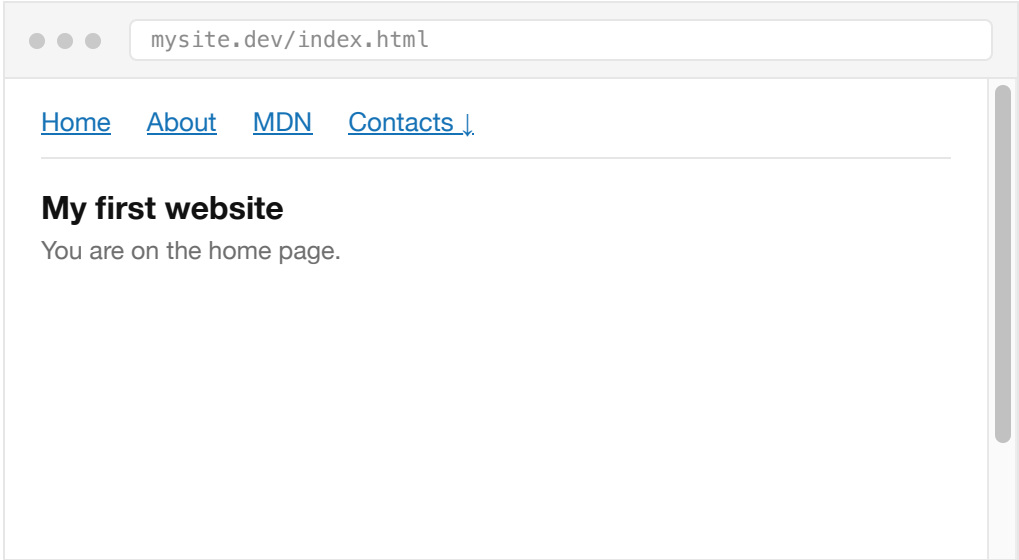
Links & navigation

nav.html

```
<nav>
  <a href="index.html">Home</a>
  <a href="pages/about.html">About</a>
  <a href="https://developer.mozilla.org"
     target="_blank" rel="noopener">MDN</a>
  <a href="#contacts">Contacts ↓</a>
</nav>

<!-- ...page content... -->

<section id="contacts">
  <h2>Contact us</h2>
</section>
```



Click a link in the mini browser — each `href` sends the browser somewhere different.

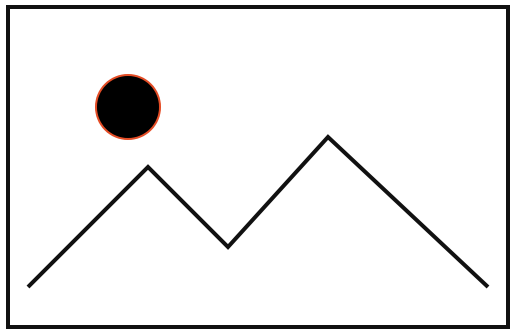
Default look: blue + underlined — CSS (section 03) restyles them. A nav bar is just a styled list of links.

Images

gallery.html

```
<figure>
  <img src="images/campus.jpg"
       alt="Students on orientation day"
       width="320" height="180"
       loading="lazy">
  <figcaption>
    Orientation day, Sept 2026
  </figcaption>
</figure>

<!-- always: folder path, alt text, -->
<!-- and explicit dimensions -->
```



Orientation day, Sept 2026 — the caption

JPG
photos, millions of colors

PNG
transparency, sharp edges

SVG
logos & icons — scales forever

WebP
modern default — smallest

Three habits: keep images in a folder (src="images/…") · ALWAYS write alt · set width & height so the page doesn't jump while loading.

Forms & inputs

signup.html

```
<label>
  Email
  <input type="email" placeholder="you@univ.edu">
</label>
<label>
  Password
  <input type="password">
</label>

<input type="checkbox" checked> Remember me
<input type="radio" name="year"> Year 1
<input type="radio" name="year"> Year 2

<label>
  Deadline <input type="date">
</label>
<textarea rows="3"></textarea>
<button>Sign up</button>
```

RENDERED CONTROLS

you@univ.edu

••••••••

Remember me Year 1 Year 2

2026-09-10 **Deadline**

Sign up

type="email" validates before JS even runs

Why type matters: the right keyboard on mobile + free validation. Wrap inputs in <label> — big tap target + screen-reader name. Same **name** on radios = one choice only.

From Word to the web

Show the Word article

your editor — `semantic.html`

type here — the preview updates as you type

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Notes That Stick</title>
</head>
<body>
  <!-- retype the article here -->

</body>
</html>
```

result — `live preview`

h1 · h2 ×2 · strong · em · ol · ul

03

CSS3

The web's presentation language: from colors and fonts to full page layouts.

Syntax | Selectors | Box model | Flexbox | Grid | Position & z-index | Responsive

CSS syntax: rules, and where they live

```
h1 { color: navy; font-size: 32px ; }
```

SELECTOR — WHICH ELEMENTS? (NEXT SLIDE)

DECLARATION = PROPERTY : VALUE ; SEMICOLON BETWEEN THEM

VALUES CARRY UNITS: 32PX, 1.6, #1572B6, 50%

Three ways to attach CSS — and the one to use:

INLINE

`style="color:navy"`

inside the HTML tag

Avoid — mixes jobs, can't be reused

INTERNAL

`<style> h1 {…} </style>`

inside <head> of the page

Fine for quick single-page demos

EXTERNAL ✓

```
<link rel="stylesheet"
      href="styles.css">
```

a separate .css file

The professional default: reuse + caching

Selectors: target anything

PATTERN	TYPE	MATCHES
`h1`	element	every <h1> on the page
`.card`	class	every element with class="card"
`#header`	id	the one element with id="header"
`nav a`	descendant	links inside <nav> — parents first, space, child
`a:hover`	pseudo-class	a link the mouse is currently over
`h1, p`	grouped	same style to h1 AND p in one rule

Class is king: default to .classes; ids only for one-off anchors; elements for global resets.

When rules fight — specificity decides:

`id — #header  100 pts`

`class — .card  10 pts`

`element — h1  1 pt`

- **Tie?** the rule written LATER wins
- **Inline style** beats everything (except !important)
- **!important** wins the fight — and your debuggability. Avoid.

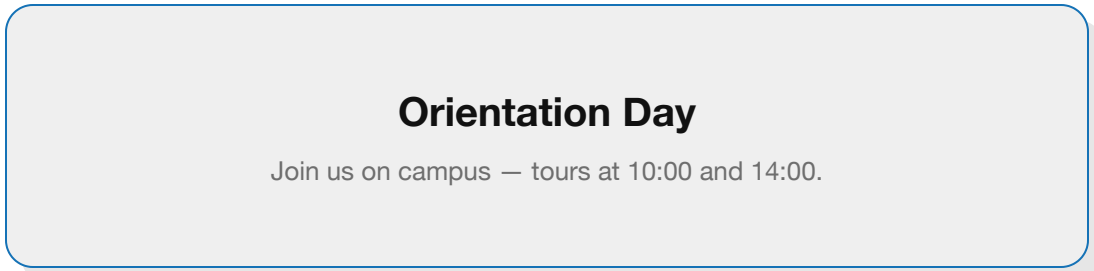
Styling: color, fonts & effects

styles.css

```css
.card {
  color: #1f2430;
  background: #eaf3fa;
  font-family: Arial, sans-serif;
  font-size: 16px;
  line-height: 1.6;
  text-align: center;
  border-radius: 16px;
  box-shadow: 8px 8px 0 #e6e6e6;
  transition: transform .3s;
}
.card:hover {
  transform: scale(1.05);
}
```

WRITING COLORS — FOUR NOTATIONS, ONE BLACK

`#111111` `rgb(17 17 17)` `rgba(17,17,17,1)` `hsl(0 0% 7%)`



hover me → grows smoothly (transition)

transition + :hover = your first animation, three lines, no JavaScript. The hover shadow above is `box-shadow: 8px 8px 0 #E6E6E6` — a hard offset, the minimalist's shadow.

Page background — your first CSS

Show the Word article

index.html styles.css

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Notes That Stick</title>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <h1>Notes That Stick</h1>
  <p>Study Skills 101 · Week 4 handout</p>

  <h2>Write it by hand</h2>
  <p>Handwriting forces your brain to
     <strong>summarize</strong>, not copy.</p>

  <h2>Review in three passes</h2>
  <ol>
    <li>Same evening — rewrite the messy parts</li>
    <li>Next day — turn headings into questions</li>
    <li>End of week — explain it to a friend</li>
```

index.html carries Lab 1's content, styles.css dresses it — the preview always shows the two files together.

Notes That Stick

Study Skills 101 · Week 4 handout

Write it by hand

Handwriting forces your brain to **summarize**, not copy.

Review in three passes

1. Same evening — rewrite the messy parts
2. Next day — turn headings into questions
3. End of week — explain it to a friend

What to capture

- **Center:** add a class to each heading, then move the text to the middle with `text-align: center`
- **Background:** change the page to a color you like — edit the `body` rule in `styles.css`

The box model: every element is a box

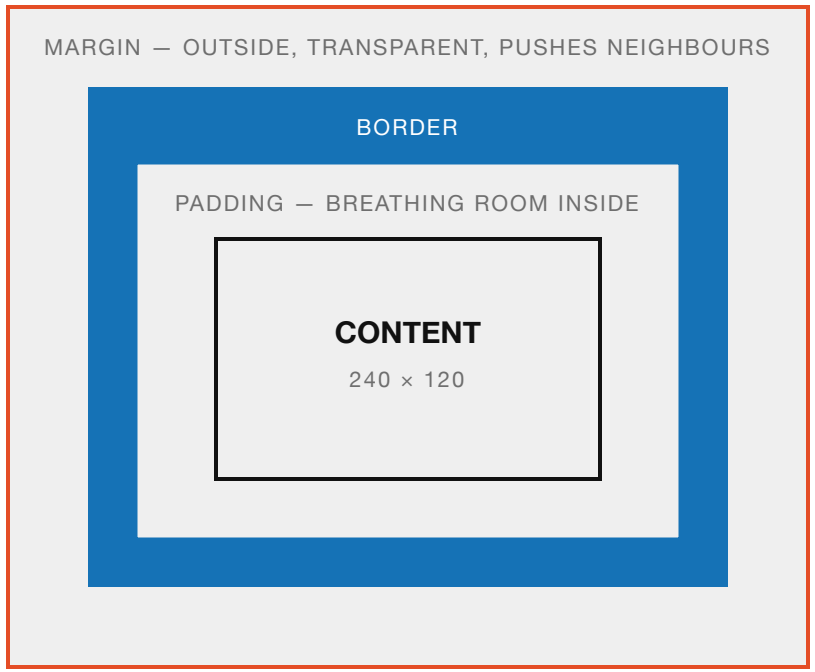


FIG. 05 — EVERY ELEMENT IS A NESTED BOX · SEE IT LIVE IN DEVTOOLS

Total width surprises beginners:

```
default width + padding + border
= wider than you asked
```

```
* { box-sizing: border-box; }
```

put this in every project — width then includes padding + border, which is what you always meant anyway.

DevTools: Inspect any element → the colored box-model diagram is drawn for you, live.

Grid: two-dimensional layout

```
gallery.css — mirrors your grid
.gallery {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  gap: 16px;
}
```

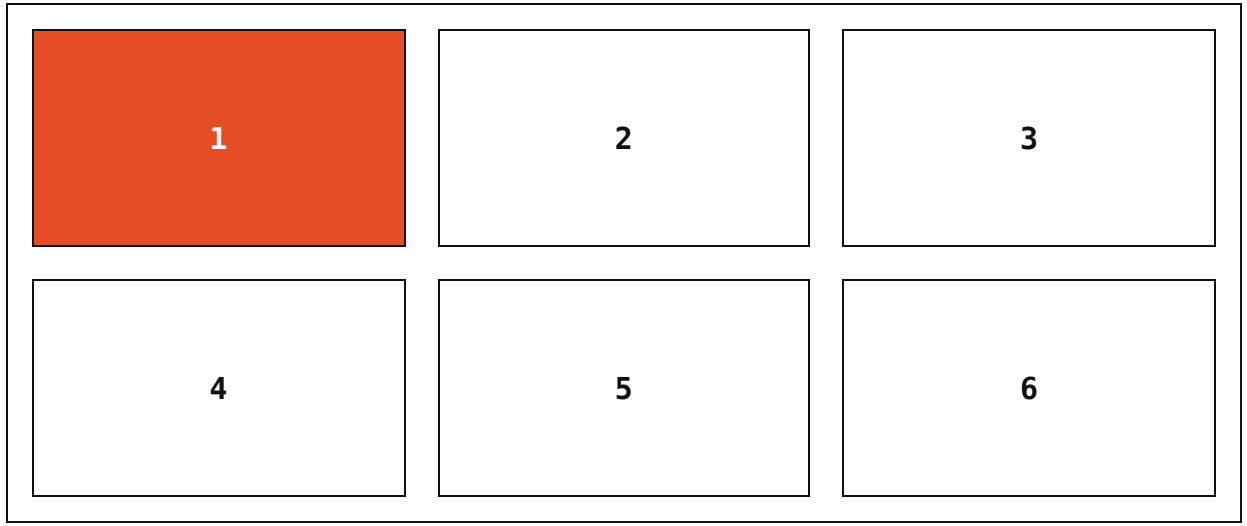
`1fr` = one fraction of the free space. Play with the controls — the code on the left is always exactly what the grid on the right is doing.

Flexbox
one row OR column · content decides size

Grid
rows AND columns · layout decides size

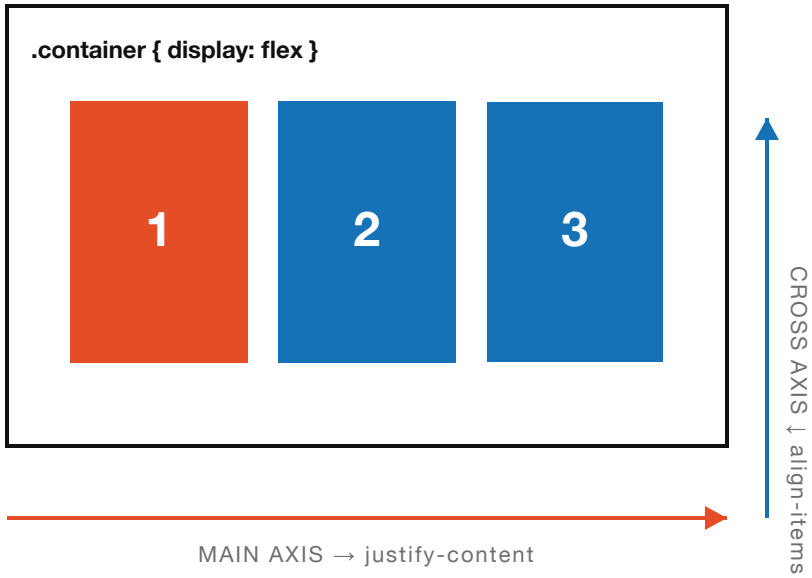
COLUMNS `repeat(2, 1fr)` **`repeat(3, 1fr)`** `repeat(4, 1fr)` `1fr 2fr`

GAP `8px` **`16px`** `32px`



click a control — six items reflow, the CSS above updates with it

Flexbox: one-dimensional layout



The five properties you actually need

`display: flex`	turns children into flex items, in a row
`flex-direction`	row (default) or column — the axis
`justify-content`	main axis: flex-start · center · space-between
`align-items`	cross axis: stretch · center · flex-end
`gap`	fixed spacing between items — no margin hacks

Position & z-index: layers and overlaps

static	default — normal flow; offset properties are ignored
relative	stays in flow, can be nudged — anchors absolute children
absolute	removed from flow; pinned to the nearest positioned ancestor
fixed	pinned to the viewport — cookie banners

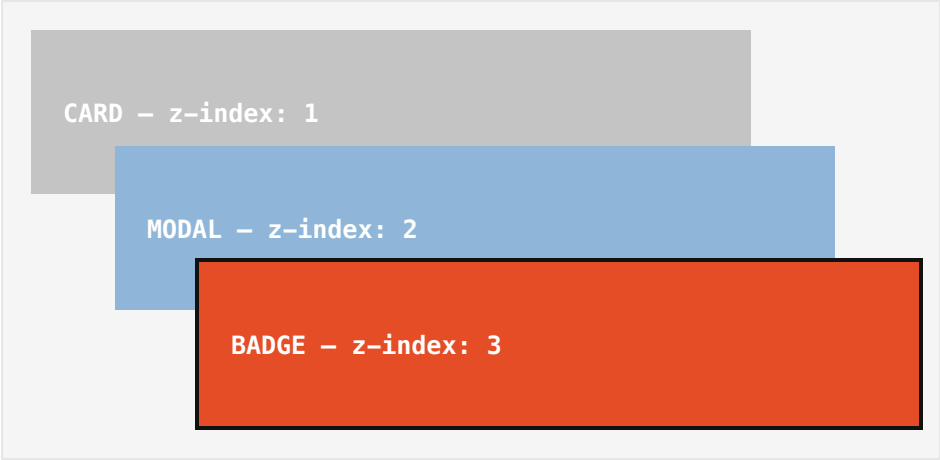
layers.css — mirrors the stack

```
.card  { position: absolute; z-index: 1 }
.modal { position: absolute; z-index: 2 }
.badge { position: absolute; z-index: 3 }   /* top */
```

Recipe: parent relative + child absolute = precise placement for badges, bubbles, stickers.

z-index decides who is on top

higher = closer to the viewer. Only positioned (or flex/grid) items stack.



Reset stack

click a layer to bring it to the front

Responsive: one page, every screen

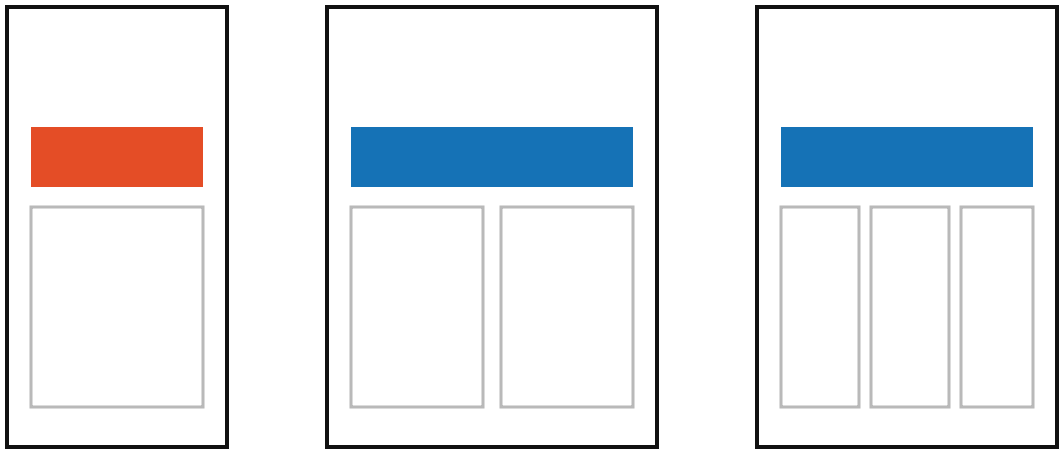
responsive.html

```
<!-- in <head>: the magic line -->
<meta
  name="viewport"
  content="width=device-width, initial-scale=1"
>
```

styles.css

```
/* adapt at breakpoints */
@media (max-width: 600px) {
  .menu {
    flex-direction: column;
  }
  h1 {
    font-size: 28px;
  }
}
```

Why: over 60% of web traffic is mobile. No viewport meta = the phone shows a zoomed-out desktop page — the #1 beginner bug.



Phone
375 px · 1 column

Tablet
768 px · 2 columns

Desktop
1280 px · 3 columns

- **Mobile-first:** style for the phone, add @media (min-width:…) to enhance
- **Fluid units:** %, rem, vw — plus max-width: 100% on images
- **Flex/Grid wrap:** let rows reflow instead of hard-coding widths

SOURCES — STATCOUNTER, DESKTOP VS. MOBILE WEB TRAFFIC

Flexbox Duck — Level 1





```
.ducks {
  display: flex;
  justify-content: ?                ;
}
```

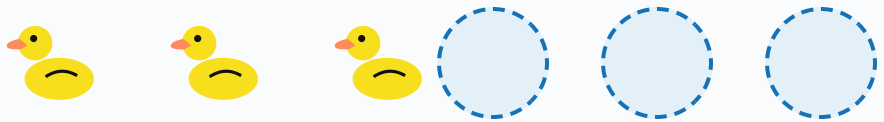
Apply ⏎ | Reset

LEVEL 1 / 5
The rings are on the RIGHT edge — move the whole row to them.

- **Type** the missing value in the code line — press Enter or Apply
- **Stuck?** start typing — a dropdown suggests matching values
- **Win:** every duck floats on its ring

Typed values are real CSS — a valid one moves the pond, a made-up one gets flagged. Experiment freely; Reset puts it back.

Flexbox Duck — Level 2



```
.ducks {
  display: flex;
  justify-content: ?                ;
}
```

Apply ⏎ | Reset

LEVEL 2 / 5
Ducks like the middle. Center the row.

- **Type** the missing value in the code line — press Enter or Apply
- **Stuck?** start typing — a dropdown suggests matching values
- **Win:** every duck floats on its ring

Typed values are real CSS — a valid one moves the pond, a made-up one gets flagged. Experiment freely; Reset puts it back.

Flexbox Duck — Level 3



```
.ducks {
  display: flex;
  justify-content: ?                ;
}
```

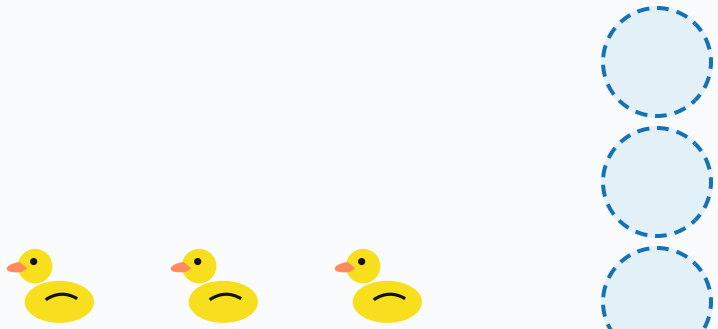
Apply ⏎ | Reset

LEVEL 3 / 5
One left, one middle, one right — spread them out.

- **Type** the missing value in the code line — press Enter or Apply
- **Stuck?** start typing — a dropdown suggests matching values
- **Win:** every duck floats on its ring

Typed values are real CSS — a valid one moves the pond, a made-up one gets flagged. Experiment freely; Reset puts it back.

Flexbox Duck — Level 4




```
.ducks {
  display: flex;
  flex-direction: ?                ;
}
```

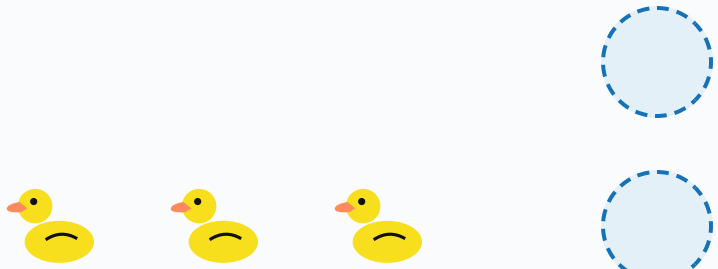
Apply ⏎ Reset

LEVEL 4 / 5
Think perpendicular — stack the row into a column.

- **Type** the missing value in the code line — press Enter or Apply
- **Stuck?** start typing — a dropdown suggests matching values
- **Win:** every duck floats on its ring

Typed values are real CSS — a valid one moves the pond, a made-up one gets flagged. Experiment freely; Reset puts it back.

Flexbox Duck — Level 5




```
.ducks {
  display: flex;
  flex-direction: ?                ;
  justify-content: ?               ;
}
```

Apply ⏎ | Reset

LEVEL 5 / 5
Stack them AND spread them from top to bottom.

- **Type** the missing value in the code line — press Enter or Apply
- **Stuck?** start typing — a dropdown suggests matching values
- **Win:** every duck floats on its ring

○ ○ ○ ○ ●

Typed values are real CSS — a valid one moves the pond, a made-up one gets flagged. Experiment freely; Reset puts it back.

04

JavaScript

The web's programming language: from variables to manipulating the page itself.

Console · Variables · Types · Math · Control flow · Loops · Arrays & objects · Functions · DOM · Fetch

What is JavaScript?

JavaScript is the browser's programming language. It reacts to people, changes the page, and can request data without a full reload.

HTML

Structure

The elements a page is made of.

CSS

Appearance

How those elements look and lay out.

JAVASCRIPT

Behaviour

What happens when a person clicks, types, or waits.

first-script.js

```
const button = document.querySelector("#save");

button.addEventListener("click", () => {
  document.body.classList.toggle("saved");
});
```

Think of it as a set of instructions that the browser runs after the HTML and CSS arrive. The next slides teach the building blocks of those instructions.

console.log(): see what your code sees

`console.log()` prints text, values, and objects in the browser's developer console. Use it to check an assumption before guessing at a bug.

console.js

```
const student = { name: "Ada", score: 10 };

console.log("Hello, web!");
console.log(student.score);   // 10
console.log(student);         // inspect the whole object
```

1. Open the page in your browser.
2. Open DevTools, then choose the **Console** tab.
3. Reload or click the page and read the output.

Debugging habit: log the value at the point it becomes surprising, then remove temporary logs when you are done.

Say it — then watch it print

hello.js — intellisense pops up while you type

```
1  // console.log prints whatever is between the parentheses
2  console.log("Hello");
3  console.log(1 + 2);
4  console.log("Points:", 10); // two arguments: label + value
5
6  // comma keeps both on one line — plus adds numbers first
7  console.log("Score:", 10 + 5);
8
9  // your turn: log something about you
10 console.log("My name is ...");
```

try: change 1 + 2 into "1" + 2 — spot the difference?

```
Hello
3
Points: 10
Score: 15
My name is ...
```

console — live output · reruns as you type

Variables: let, const, var

KEYWORD	REASSIGN?	REDECLARE?	SCOPE	WHEN TO USE
`const`	no	no	block	**DEFAULT choice** — values that never change
`let`	yes	no	block	values that WILL be reassigned (score, counters)
`var`	yes	yes	function	legacy 1995–2015 — you'll meet it in old code; don't write it

Rule of thumb: start with const. Switch to let the moment you need to reassign. Pretend var doesn't exist.

```
1995: var

2015:
let, const
```

variables.js

```
const courseName = "Web Development";   // fixed
let score = 0;                          // will change
score = score + 10;                     // now 10
```

var, let, const — who survives?

variables.js — run it, then uncomment the trap lines

```
// three ways to declare — meet var, let, const
var one = "One";
let two = "Two";
const three = "Three";

one = "1"; // var — reassign, no complaint
two = "2"; // let — reassign, no complaint
// three = "3"; // ← uncomment me: const refuses

console.log(one, two, three);

// block scope: let stays inside the curly braces
{
  var insideVar = "var leaks out of the block";
  let insideLet = "let stays in the block";
}
console.log(insideVar); // works
// console.log(insideLet); // ← uncomment me: ReferenceError
```

try: uncomment both trap lines — the console shows real errors in red.

```
1 2 Three
var leaks out of the block
```

console — live output · reruns as you type

Data types & typeof

7 primitives …

string	`"Ada"`	text in quotes — or backticks
number	`42 · 3.14`	integers and decimals, one type
boolean	`true · false`	the two logic values
undefined	`not set`	"no value yet"
null	`empty`	"we cleared it on purpose"
bigint	`9007…993n`	astronomically big integers
symbol	`Symbol("id")`	rare — unique identifiers

Strings with superpowers: backtick template literals: `` `Total: ${price} baht` `` — interpolation without + concatenation.

… plus one composite:

typeof.js

```
console.log(typeof "hello");    // string
console.log(typeof 42);         // number
console.log(typeof true);       // boolean
console.log(typeof undefined);  // undefined
console.log(typeof null);       // object !
console.log(typeof [1, 2, 3]);  // object
console.log(typeof {a: 1});     // object

// objects bundle many values:
const student = { name: "Ada" };
```

Historic bug: typeof null says "object" — a 1995 type-tag mistake kept so old sites don't break. null is NOT an object.

Six types, one console

types.js — ask every value what it is with typeof

```
1  // string
2  let name = "Somchai";
3  console.log(`Hello, ${name}`); // template literal
4  console.log(typeof name); // "string"
5
6  // number
7  let price = 99.5;
8  console.log(price * 2); // 199
9
10 // boolean
11 let isStudent = true;
12 console.log(typeof isStudent); // "boolean"
13
14 // object — data in key: value pairs
15 let person = { name: "Somying", age: 30 };
16 console.log(person.name); // dot access
17 console.log(person["age"]); // bracket access
18
19 // array — an ordered list
20 let fruits = ["apple", "banana", "orange"];
21 console.log(fruits[0]); // "apple"
22 console.log(fruits.length); // 3
```

try: add a `hobbies` array to `person`, then log `person.hobbies[0]`.

```
Hello, Somchai
string
199
boolean
Somying
30
apple
3
```

console — live output · reruns as you type

Math operations & operators

Operators you'll use daily

`+ - * /`	the classics: 7 * 6 → 42
`%`	remainder: 7 % 2 → 1 — even/odd, cycles
`**`	power: 2 ** 10 → 1024
`+= -= *= ++`	shortcuts: score += 10 ≡ score = score + 10
`+ on strings`	"5" + 2 → "52" — JS glues instead of adds!

Watch out: input fields return STRINGS. Convert before math: `Number(input.value)` or `parseInt("42px")` → 42.

math.js

```
const price = 19.99, qty = 3;
const total = price * qty;      // 59.97

total.toFixed(2);               // "59.97"

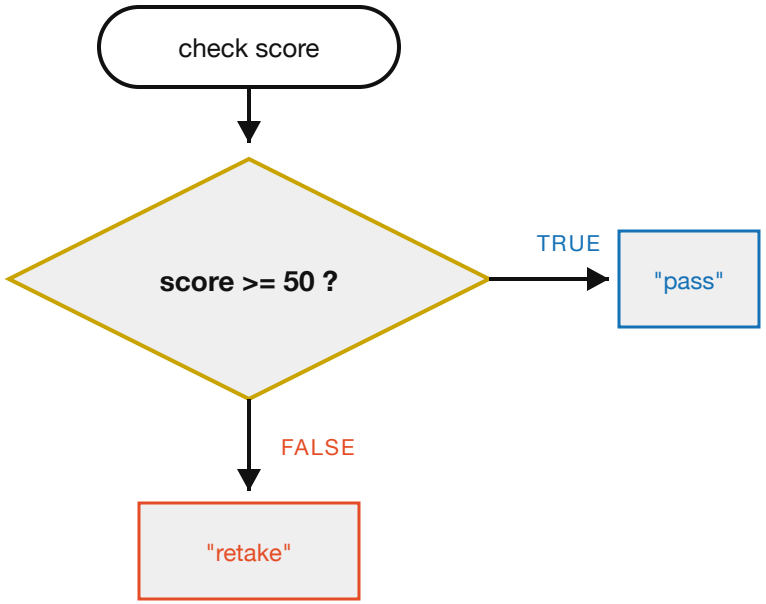
Math.round(4.6);                // 5
Math.floor(4.9);                // 4
Math.ceil(4.1);                 // 5

// random die: 1..6
Math.floor(Math.random() * 6) + 1;

const msg = `Total: ${total}`;  // template
```

Why it matters: a calculator page is 90% of this slide — read strings, convert, compute, display.

Control flow: making decisions



THE DIAMOND IS THE IF — CODE BRANCHES EXACTLY LIKE THE DIAGRAM

```
grades.js
let grade;
if (score >= 50) {
  grade = "pass";
} else if (score >= 40) {
  grade = "retake";
} else {
  grade = "fail";
}

// many exact values? use switch:
let off;
switch (day) {
  case "Sat":
  case "Sun": off = true; break;
  default: off = false;
}
```

`===` `==` `!==` `&& || !` `< > <= >=`

Truthy / falsy: 0, "", null, undefined, NaN behave like false; everything else like true. And remember: **===** strict beats **==** loose ("5" == 5 is true, "5" === 5 is false).

Decisions: pass or fail?

decisions.js — change the numbers, watch the branch flip

```
// if / else — pick one branch
let score = 45;
if (score >= 50) {
  console.log("pass");
} else {
  console.log("fail"); // ← this one runs
}

// else if — checks in order, stops at the first hit
let grade = 75;
if (grade >= 80) console.log("grade A");
else if (grade >= 70) console.log("grade B"); // ← runs
else console.log("grade F");

// && means both, || means either
let username = "admin";
let password = "1234";
if (username === "admin" && password === "1234") {
  console.log("login ok");
} else {
  console.log("wrong username or password");
}
```

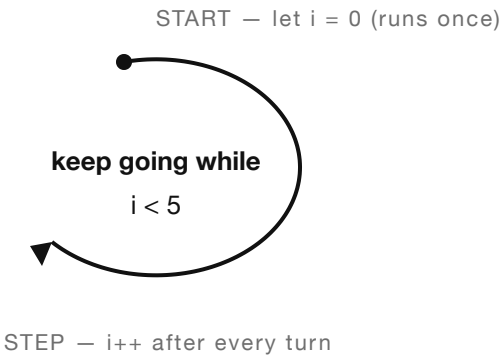
try: set `score` to 80 and `password` to "0000" — which lines run now?

```
fail
grade B
login ok
```

console — live output · reruns as you type

Loops: repeat without repeating yourself

```
for ( let i = 0; i < 5; i++ ) { console.log(i); }
```





press Run — i starts at 0, one chip per turn

```
loops.js
// counted: 0, 1, 2, 3, 4
for (let i = 0; i < 5; i++) {
  console.log("Cube " + i);
}

// walk an array: for...of
for (const c of colors) show(c);
```

for	you know how many times
while	repeat UNTIL something happens
break / continue	exit early / skip one turn

Why loops matter: one loop stamps 100 cubes onto the page — imagine copy-pasting 100 divs by hand instead.

Four ways to loop

loops.js — same list, four different walks through it

```
let fruits = ["apple", "banana", "orange"];

// classic for — you control the index
for (let i = 0; i < fruits.length; i++) {
  console.log(i, fruits[i]);
}

// for...of — the values, no index
for (let fruit of fruits) {
  console.log(fruit);
}

// forEach — array method with a callback
fruits.forEach((fruit, index) => {
  console.log(`${index}: ${fruit}`);
});

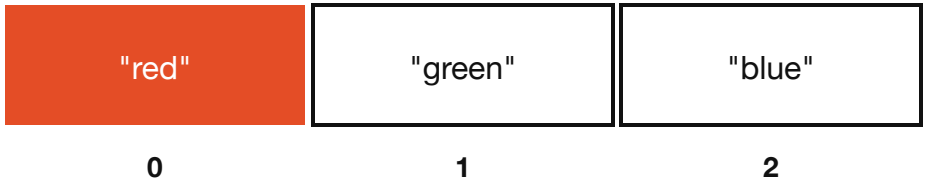
// map — build a NEW array from the old one
let numbers = [1, 2, 3, 4];
let doubled = numbers.map((n) => n * 2);
console.log(doubled); // [2, 4, 6, 8]
```

try: add `"mango"` to the list — predict all three loops before you look.

```
0 apple
1 banana
2 orange
apple
banana
orange
0: apple
1: banana
2: orange
[2,4,6,8]
```

console — live output · reruns as you type

Arrays: ordered collections



colors[0] is the FIRST item — counting starts at 0, the #1 beginner surprise.

colors.length → 3

arrays.js

```
const colors = ["red", "green", "blue"];

colors[0];              // "red"
colors.length;          // 3
colors[1] = "lime";     // replace

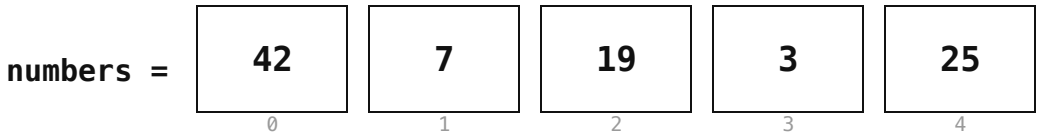
colors.push("gold");    // add to end
colors.pop();           // remove from end
colors.includes("red") // true

colors.join(" · ");     // "red · lime · blue"
```

- **Ordered:** items keep their position — use [index] to reach any of them
- **Mixed:** one array may hold strings, numbers, even other arrays
- **const arrays mutate:** const stops REASSIGNMENT, not push/pop
- **Loop over it:** for (const c of colors) — every item, in order

Recipe to remember: array of settings + one loop = many DOM elements, zero copy-paste.

Array methods, visualized



forEach | map ×2 | filter > 10 | push(99) | pop | reset

```
arrays.js — mirrors the buttons
numbers = [42, 7, 19, 3, 25]
```

press a method — watch what happens to the array

- **`forEach`** just visits every item — for side effects; returns nothing
- **`map`** transforms every item → a **new** array, same length
- **`filter`** keeps the items that pass the test → a **new** array
- **`push` / `pop`** add / remove at the end — these change the original

Remember: map & filter leave the original untouched — that is the habit modern JS (and React) is built on.

Objects: describing real things

const student = { … }

name	"Ada"
year	2
gpa	3.86
skills	["html","css"]
active	true

A KEY DESCRIBES A PROPERTY — LIKE A LABELED DRAWER

objects.js

```
const student = {
  name: "Ada", year: 2, gpa: 3.86,
  skills: ["html", "css"],
};

student.name;          // "Ada"  (dot)
student["gpa"];        // 3.86   (bracket)
student.skills.push("js");

// arrays of objects = a mini database:
const students = [
  { name: "Ada",  year: 2 },
  { name: "Bo",   year: 3 },
];
students[1].name;      // "Bo"
```

The big idea: one object describes one real thing — name, year, gpa. Real pages render arrays of these; change the data and the page follows — exactly how React props feel later.

Functions: reusable recipes

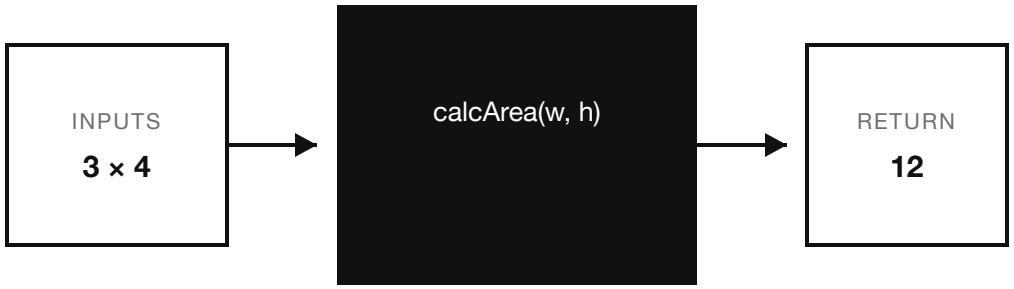
functions.js

```
// declaration: name + parameters
function calcArea(width, height) {
  const area = width * height;
  return area;        // hand back
}

// call it with arguments:
const a = calcArea(3, 4);   // 12
const b = calcArea(10, 2);  // 20

// arrow function — same idea, shorter:
const calcArea2 = (w, h) => w * h;
calcArea2(5, 5);            // 25
```

On a calculator page, every key calls the SAME function with a different argument.



PARAMETERS (W, H) ARE PLACEHOLDERS — ARGUMENTS (3, 4) FILL THEM IN

Why functions: write once, call anywhere — the calculator's compute() runs on every button press. No return means undefined.

Write once, call twice

functions.js — the TODO is yours, the console is the judge

```
// declaration — a recipe you can reuse
function greet(name) {
  return `Hello, ${name}`;
}
console.log(greet("Somchai"));
console.log(greet("Somying"));

// arrow function — same idea, shorter
const shout = (text) => text.toUpperCase();
console.log(shout("quiet please"));

// your turn: write double(n) that returns n * 2,
// then log double(21) on the line below
```

try: log `double(21)` first — the console shows the error until `double` exists.

```
Hello, Somchai
Hello, Somying
QUIET PLEASE
```

console — live output · reruns as you type

The DOM: JavaScript meets your HTML

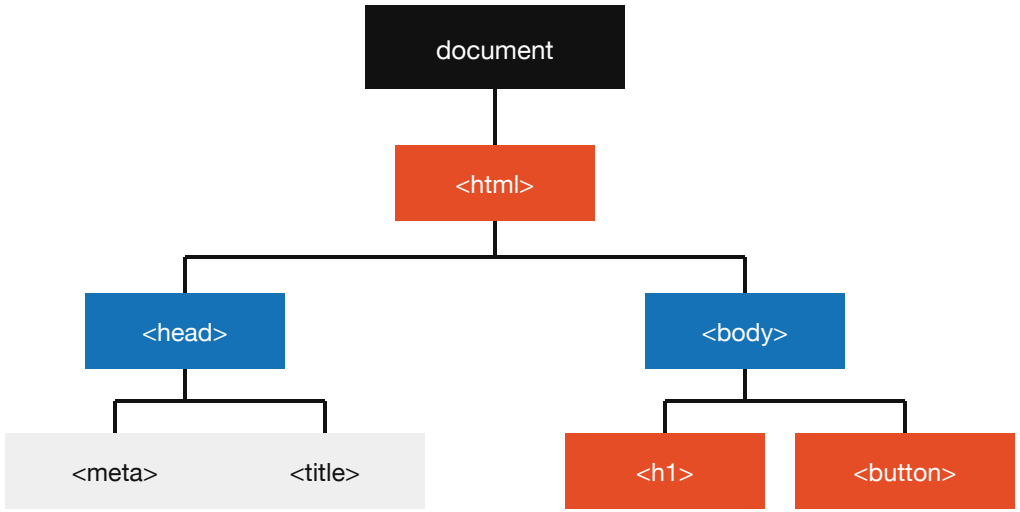


FIG. 06 — THE BROWSER READS YOUR HTML INTO THIS LIVING TREE

1 select 2 listen 3 change 4 create — *behind every interactive page*

dom.js

```
// 1  select the button
const btn = document.querySelector("#go");

// 2  listen for events
btn.addEventListener("click", () => {
  // 3  change what is on screen
  const title = document.querySelector("h1");
  title.classList.toggle("done");

  // 4  create & attach new content
  const p = document.createElement("p");
  p.textContent = "Clicked!";
  document.body.append(p);
});
```

The user directory = all four moves — fetch the data, then build one element per user

Fetch API: ask a server for data

`fetch()` is built into JavaScript. It makes an HTTP request, waits for the response, then lets your page use the data — here, a free test API that returns 10 fake users.

users.js

```
async function loadUsers() {
  const response = await fetch(
    "https://jsonplaceholder.typicode.com/users");
  if (!response.ok) throw new Error("Request failed");

  const users = await response.json();
  console.log(users);
}

loadUsers().catch(console.error);
```

each user: { name, email, address: { street, city, … } }

1 · REQUEST

Ask for the URL with `fetch()`.

2 · RESPONSE

Check whether the server accepted the request.

3 · DATA

Read the JSON — an array of 10 user objects, ready to render.

Reshape an API response

- **fetch** the users array — same API as the fetch-api slide
- **map** every user to an object with exactly two keys: `name` and `address`
- **address** = street, suite, city and zipcode joined into one string

users.js — one .map() is all it takes

```
async function getUsers() {
  const res = await fetch(
    "https://jsonplaceholder.typicode.com/users");
  const users = await res.json();

  // TODO: replace this return — use .map() to build
  // [{ name, address }] like the example on the right
  return users;
}

getUsers().then((users) => console.log(users));
```

stuck? `console.log(users[0])` first and look at the shape of one user.


```
[{"id":1,"name":"Leanne
Graham","username":"Bret","email":"Sincere@april.bi
z","address":{"street":"Kulas Light","suite":"Apt.
556","city":"Gwenborough","zipcode":"92998-
3874","geo":
{"lat":"-37.3159","lng":"81.1496"}},"phone":"1-770-
736-8031
x56442","website":"hildegard.org","company":
{"name":"Romaguera-Crona","catchPhrase":"Multi-
layered client-server neural-net","bs":"harness
real-time e-markets"}},{"id":2,"name":"Ervin
Howell","username":"Antonette","email":"Shanna@meli
ssa.tv","address":{"street":"Victor
Plains","suite":"Suite
879","city":"Wisokyburgh","zipcode":"90566-
7771","geo":
{"lat":"-43.9509","lng":"-34.4618"}},"phone":"010-
```


console — live output · reruns as you type

expected: [{ name: "Leanne Graham", address: "Kulas Light, Apt. 556, Gwenborough 92998-3874" }, …]

A user directory from a live API

Show the task

index.html | style.css | script.js

```
 1 <!DOCTYPE html>
 2 <html lang="en">
 3   <head>
 4     <meta charset="UTF-8">
 5     <title>User directory</title>
 6   </head>
 7   <body>
 8     <h1>User directory</h1>
 9     <!-- one <li> per user is appended here by script.js -->
10     <ul id="user-list"></ul>
11   </body>
12 </html>
```

preview — `index.html + style.css`

User directory

console — `script.js`

console — output shows up here

your move: finish the TODO in `script.js` — the preview fills with ten real users, no page reload. Stuck? `console.log(users[0])` first.

WEB DEVELOPMENT ACADEMY

You can now build the web.

Structure it with HTML. Style it with CSS. Make it think with JavaScript.
Then practice until it feels like handwriting.

KEEP THESE OPEN FOREVER

MDN Web Docs
developer.mozilla.org

Flexbox Froggy
flexboxfroggy.com — all 24 levels

freeCodeCamp
freecodecamp.org — the full free curriculum

Can I use
caniuse.com — browser support

Questions?