

HTML & CSS Essentials

Semantic HTML, the Box Model, Flexbox, Grid & Responsive — the static foundation

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-14

Abstract The static foundation of the presentation layer. You refresh semantic HTML (document structure, landmarks, text, lists, links, images, tables, forms, SEO, accessibility) and CSS fundamentals (the box model, selectors, the cascade and specificity, Flexbox, Grid, responsive / mobile-first). Every example builds one artifact: a fully static review card — one featured card laid out with Flexbox and a responsive grid of review cards in CSS Grid — in plain HTML and CSS. No scripting, no framework. The app’s real styling approach (Tailwind 3.4) is referenced throughout.

Table of contents

1	What this session covers	1	3.3	13. The cascade — why one rule won	12
2	Part A — HTML: structure and meaning .	2	3.4	14. Layout with Flexbox	13
2.1	1. Document skeleton & doctype	2	3.5	15. Layout with CSS Grid	14
2.2	2. Semantic landmarks	3	3.6	16. Responsive design — media queries & fluid units, mobile-first . .	15
2.3	3. Heading hierarchy & text semantics	4	3.7	17. Tailwind CSS 3.4 — the app’s real styling approach	16
2.4	4. Lists	5	4	Key takeaways	17
2.5	5. Links & internal anchors	5			
2.6	6. Images	6			
2.7	7. Tables	6			
2.8	8. Forms (brief)	7			
2.9	9. SEO basics	7			
2.10	10. Accessibility basics	8			
3	Part B — CSS: style and layout	10			
3.1	11. The box model	10			
3.2	12. Selectors & specificity	11			

1 What this session covers

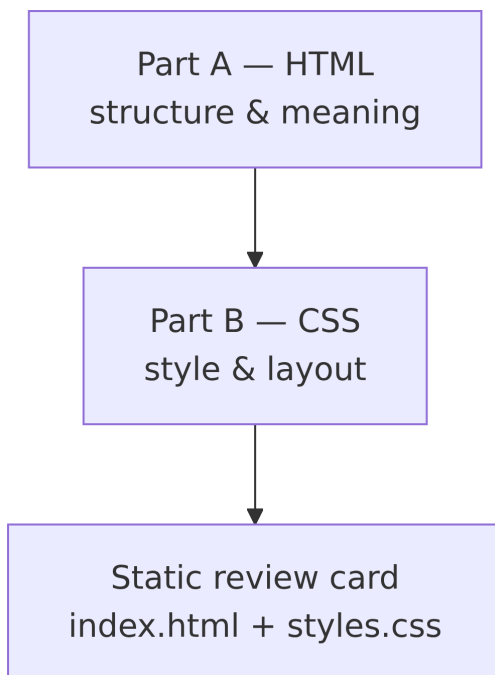
This session is the **static foundation** of the presentation layer: the HTML and CSS you need before any interactivity. There is **no JS** and no framework here — the page is built from two plain documents and one stylesheet, and you can read every rule.

The threaded artifact. Across all of Part A and Part B you build one thing: a **static review card** — one page (`index.html`) with a featured review card laid out with Flexbox and a responsive grid of review cards in CSS Grid, plus one shared stylesheet (`styles.css`). Each section adds a piece of it, so by the end the page is complete, semantic, accessible, and responsive.

The product context (name, image, price) is **given** from the app's existing product catalog — you are not building the catalog. You are building the first piece of the review layer that students will extend across S1–S5.

💡 Where this fits

This is the static floor of the course. In **Session 3** the review card becomes an interactive React component with a review form; the markup and layout you build here carry straight over. You are laying the foundation, not the interactivity.



The files you will build:

File	Role
<code>index.html</code>	The reviews page — a featured review card + a responsive grid of review cards.
<code>styles.css</code>	The single shared stylesheet.

2 Part A — HTML: structure and meaning

HTML (**HyperText Markup Language**) is the standard language for web pages. It defines **what the content is** — its structure and meaning — not how it looks (that is CSS) and not how it behaves (that is scripting, later). An HTML document is a tree of nested **elements**, each written with a start tag, optional content, and an end tag.

2.1 1. Document skeleton & doc-type

Every page starts with the same skeleton. The `<!DOCTYPE html>` line tells the browser to parse the document as modern HTML5; without it, browsers fall back to quirks mode. `<html>` is the root and its `lang` attribute sets the document language (used by screen readers and translation). `<head>` holds metadata that is not rendered; `<body>` holds everything that is.

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport"
content="width=device-width,
initial-scale=1.0">
    <meta name="description"
  
```

```

content="Customer reviews for the
Wireless Headphones.">
    <title>Reviews - Wireless
Headphones | Acme</title>
    <link rel="stylesheet"
href="styles.css">
</head>
<body>
    <!-- rendered content goes here
-->
</body>
</html>

```

- `<meta charset="UTF-8">` — the character encoding; keep it first in `<head>`.
- `<meta name="viewport" ...>` — **required** for responsive design; it tells mobile browsers to use the device width instead of a virtual desktop width.
- `<link rel="stylesheet">` — attaches your external CSS. (You *link* a stylesheet; you do not embed it inline unless the page is throw-away.)

2.2 2. Semantic landmarks

A **landmark** is an element that names a region of the page. Screen readers let a user jump straight to “main content” or “navigation”; search engines use the regions to understand the page. The set you will always reach for:

Element	Names	Rule of thumb
<code><header></code>	Site / section header	Branding + top navigation.
<code><nav></code>	A block of links	Wrap the main link set; label it if there is more than one.

Element	Names	Rule of thumb
<code><main></code>	The dominant content	Exactly one per page.
<code><section></code>	A thematic group	Has its own heading; a chunk of the main content.
<code><article></code>	A self-contained item	One product card, one review — meaningful on its own.
<code><aside></code>	Tangential content	“Related products”, a sidebar note.
<code><footer></code>	Page / section footer	Copyright, contact, secondary links.

Worked example — the reviews page skeleton. This is the structural frame of `index.html` before we style it. One `<main>`, one `<nav>`, and each review card is an `<article>` because a review makes sense on its own.

```

<body>
  <header>
    <h1>Reviews - Wireless
Headphones</h1>
    <nav aria-label="Primary">
      <ul>
        <li><a href="index.html" aria-
current="page">Reviews</a></li>
      </ul>
    </nav>
  </header>

```

```

<main>
  <section class="review-grid" aria-label="All reviews">
    <article class="review-card">...
  </article>
    <article class="review-card">...
  </article>
    <article class="review-card">...
  </article>
  </section>
</main>

<footer>
  <p>&copy; 2026 Acme. All rights reserved.</p>
</footer>
</body>

```

i Note

<div> vs a landmark. Use a landmark when a region has meaning; use `<div>` (a neutral grouping box with no meaning) only when you need a hook for layout or styling. The `review-grid` above is a `<section>` because “all reviews for this product” is a thematic grouping with its own implicit heading — a content concern, not just layout.

2.3 3. Heading hierarchy & text semantics

Headings describe the outline of the page. There are six levels, `<h1>`–`<h6>`. The rules that matter:

- **One `<h1>`** per page — the page’s main title.
- **No skipped levels** — go `h1` → `h2` → `h3`, never `h1` → `h3`. A screen-reader user navigates by heading level; a jump breaks the outline.
- Choose a level by **outline position**, not by the size you want. (Size is CSS, not HTML)

— never pick `<h2>` just because you want bigger text.)

Inline text semantics mark the meaning of a span of text:

Element	Meaning	Example in the review card
<code><p></code>	A paragraph of prose.	The review body text.
<code></code>	A neutral in-line grouping (no meaning).	The source badge text.
<code></code>	Strong importance (bold by default).	“Verified purchase” .
<code></code>	Stress / emphasis (italic by default).	“Ships tomorrow” in a review.
<code><mark></code>	Relevance / highlight.	The matched search term in a review body.
<code><time></code>	A machine-readable date.	<code>created_at</code> on a review.

```

<article class="review-card">
  <header class="review-card-header">
    <span class="review-source">Amazon</span>
    <span class="review-author">M. Dubois</span>
    <span class="review-rating">5/5</span>
  </header>
  <h2>Exceeded expectations</h2>

```

```

    <p>Battery lasts all day,
    <strong>verified purchase</strong>,
        noise cancelling is
    <em>impressive</em>.</p>
    <footer class="review-card-footer">
        <time
datetime="2026-03-14">2026-03-14</
time>
        <span class="review-verified">✓
Verified purchase</span>
    </footer>
</article>

```

2.4 4. Lists

HTML has three list types. A list is a container (, , or <dl>) holding items.

- **Unordered list** — a set of items, order does not matter (bulleted).
- **Ordered list** — a sequence, order matters (numbered).
- **Definition list** <dl> — term / description pairs (glossaries, specs).

```

<!-- Unordered: a product's features
(no fixed order) -->
<ul class="features">
    <li>30-hour battery</li>
    <li>Active noise cancelling</li>
    <li>Bluetooth 5.3</li>
</ul>

<!-- Ordered: how to assemble it
(order matters) -->
<ol class="assembly">
    <li>Unbox the headset</li>
    <li>Charge for 2 hours</li>
    <li>Pair over Bluetooth</li>
</ol>

<!-- Definition list: the specs block
on the detail page -->
<dl class="specs">

```

```

    <dt>Driver</dt>
    <dd>40 mm dynamic</dd>
    <dt>Weight</dt>
    <dd>250 g</dd>
    <dt>Warranty</dt>
    <dd>2 years</dd>
</dl>

```

A <dl> is the right tool for the **product-specs** block on `product.html`: each <dt> is a spec name, each <dd> its value. It is more honest than a table here, because a spec block is a set of label/value pairs, not a two-dimensional grid of data.

2.5 5. Links & internal anchors

The link <a> is the backbone of a multi-page site. Its `href` is the destination:

- **Relative** (`product.html`) — resolves against the current page's folder. Use these for links inside your own site.
- **Anchor** (`#reviews`) — jumps to an element with that `id` on the same page.
- **Absolute** (`https://...`) — a full URL, for external sites.

Worked example — in-page anchors on the reviews page. On the reviews page, in-page anchors let a user jump to the featured review or the full grid without scrolling.

```

<!-- On index.html: in-page anchors
-->
<h2 id="featured">Featured review</h2>
<h2 id="all-reviews">All reviews</h2>

<nav aria-label="On this page">
    <a href="#featured">Featured</a> ·
    <a href="#all-reviews">All reviews</a>
</nav>

```

```
<!-- A review card with an anchor for
deep-linking -->
<article      class="review-card"
id="review-1">
  <header      class="review-card-
header">...</header>
  <h2>Exceeded expectations</h2>
  ...
</article>
```

i Note

aria-current="page" marks the nav link for the page you are on — a small accessibility and UX win that costs one attribute.

2.6 6. Images

`<img>` is a **void** element (no closing tag). Three attributes matter:

- **src** — the image's URL.
- **alt** — **alternative text**. Read by screen readers when the image cannot be seen; shown if the image fails to load. Describe the image's *meaning*, not "image of...".
- **width / height** — the intrinsic size in pixels. Setting them **prevents layout shift**: the browser reserves the box before the image loads, so the page does not jump.

```

```

- **Meaningful image** → a descriptive **alt**. (The headphones photo above.)
- **Decorative image** → an empty **alt=""** so screen readers skip it.

- **Image that carries content** (a chart, a diagram) → **alt** that conveys the data, or a real text equivalent nearby.

2.7 7. Tables

A **table** presents two-dimensional data: rows of related values. It is the right element when the data genuinely has columns and rows (a comparison, a schedule, a price list) — not as a layout trick. The structure:

- `<table>` wraps the whole thing.
- `<thead>` holds the header row; `<tbody>` holds the data rows.
- `<tr>` is a row; `<th>` a header cell (bold, centered by default, scoped); `<td>` a data cell.

Worked example — a product-specs comparison table. On `product.html`, a table compares two finishes of the same product:

```
<table class="specs-table">
  <caption>Finishes for the Wireless
Headphones</caption>
  <thead>
    <tr>
      <th scope="col">Attribute</th>
      <th scope="col">Matte Black</th>
      <th scope="col">Silver</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th scope="row">Price</th>
      <td>$199.99</td>
      <td>$209.99</td>
    </tr>
    <tr>
      <th scope="row">In stock</th>
      <td>Yes</td>
      <td>Pre-order</td>
    </tr>
  </tbody>
</table>
```

`<caption>` gives the table a title (crucial for screen readers). `scope="col" / scope="row"` tells assistive technology which cells a header describes. `colspan` and `rowspan` let a cell span multiple columns or rows when the data requires it.

2.8 8. Forms (brief)

A **form** collects input. You will not wire one up in this static session — there is no server and no scripting to submit it — but you need the markup, because in **Session 3** the review card becomes an interactive React component that includes an “**add a review**” form.

The three rules:

1. **Every input has a `<label>`**, tied by the label’s `for` to the input’s `id`. Clicking the label focuses the field; screen readers announce it.
2. **Pick the right `type`** — the browser gives you the matching keyboard, validation, and (on mobile) input panel for free.
3. **Use `required` and `name`** — `name` is what gets sent; `required` is basic client-side validation.

```
<!-- The form arrives in Session
3 (React). Shown here for markup
reference. -->
<form action="#" method="post"
class="review-form">
  <h2>Add a review</h2>

  <label for="author">Name</label>
    <input type="text" id="author"
name="author" required>

  <label for="rating">Rating (1–5)</
label>
    <input type="number" id="rating"
name="rating" min="1" max="5"
required>
```

```
<label for="title">Title</label>
  <input type="text" id="title"
name="title" required>

  <label for="body">Your review</
label>
  <textarea id="body" name="body"
rows="4" required></textarea>

  <button type="submit">Submit
review</button>
</form>
```

Common type values you will meet: `text`, `email`, `number`, `date`, `search`, `checkbox`, `radio`, `range`. A `<button type="submit">` is the real submit control — and note that a `<button>` inside a form defaults to `type="submit"` when no type is given, which surprises people.

i Note

The S1 deliverable is the review card, not the form. The card is a *display* artifact: it presents one review (source, author, integer rating, title, body, date, verified status). The *input* artifact — the form that creates a new review — arrives in Session 3 as a React component. You are building the read side first.

2.9 9. SEO basics

Search engines read the **HTML you ship**. The static page is exactly where good SEO lives, because the content is present in the initial document — no scripting needed to make it visible. The levers, in order of impact:

1. **`<title>`** — the page title (the browser tab and the search-result headline). One per page, specific, ~60 characters.

2. **<meta name="description">** — the search-result snippet. One per page, a real sentence, ~150 characters.
3. **Semantic structure** — one **<h1>**, a clean heading outline, landmarks. It is how the engine maps the page's meaning.
4. **Meaningful alt text** — images are indexed through their **alt**.
5. **Readable URLs** — `product.html` beats `p?id=42`.

The reviews page gets its own **<title>** and description:

```
<!-- index.html -->
<title>Reviews - Wireless Headphones
| Acme</title>
<meta      name="description"
content="Read customer reviews for
the Wireless Headphones: ratings,
verified purchases, and helpful
feedback from Amazon, BestBuy, and
Walmart.">
```

2.10 10. Accessibility basics

Accessibility (**a11y**) means the page works for everyone — including people who use a screen reader, a keyboard only, low vision, or a slow connection. Most of it falls out of **semantic HTML done well**; the rest is a handful of habits.

- **Alt text** on every meaningful image (section 6).
- **Labels** on every form control (section 8) — never a placeholder as the only label.
- **Landmarks** (section 2) — they are the user's way to jump around.
- **Keyboard** — every interactive element must be reachable and operable by keyboard. Real **<a>**, **<button>**, and form controls are focusable for free; a styled **<div>** is not (and with no scripting here, you are safe by default).

- **Contrast** — body text needs a contrast ratio of at least **4.5:1** against its background; large text, 3:1. Check with a contrast checker before you ship a palette.
- **Focus is visible** — do not remove the focus outline without replacing it.

💡 The cheapest accessibility wins

Semantic elements + **alt** + **label** + a sane heading outline get you most of the way with zero extra code. That is the point of doing Part A properly: **a11y is a consequence of semantics, not a separate task**.

The reviews page, assembled. The pieces from sections 1–7 fit together into the complete `index.html`. This is the target for Part A — build it top to bottom, then move on to styling it in Part B.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport"
content="width=device-width,
initial-scale=1.0">
    <meta name="description"
content="Read customer reviews for
the Wireless Headphones.">
    <title>Reviews - Wireless
Headphones | Acme</title>
    <link rel="stylesheet"
href="styles.css">
  </head>
  <body>
    <header>
      <h1>Reviews for Wireless
Headphones</h1>
      <p class="product-context">
        
    Wireless Headphones – $199.99
</p>
</header>

<main>
    <!-- Featured review card
    (Flexbox) -->
    <article class="review-card
    featured">
        <header class="review-card-
        header">
            <span class="review-
            source">Amazon</span>
            <span class="review-
            author">M. Dubois</span>
            <span class="review-
            rating">5/5</span>
        </header>
        <h2>Exceeded expectations</h2>
        <p>Battery lasts all day, sound
        is crisp and the noise cancelling is
        impressive for the price.</p>
        <footer class="review-card-
        footer">
            <time
            datetime="2026-03-14">2026-03-14</
            time>
            <span class="review-
            verified">✓ Verified purchase</span>
        </footer>
    </article>

    <!-- Review grid (CSS Grid) -->
    <section class="review-grid"
    aria-label="All reviews">
        <article class="review-card">
            <header class="review-
            card-header">
                <span class="review-
                source">BestBuy</span>
                <span class="review-

```

```

author">A. Chen</span>
            <span class="review-
            rating">4/5</span>
        </header>
        <h2>Great for the price</h2>
        <p>Solid build, slight
        rattle at high volume but overall good
        value.</p>
        <footer class="review-
        card-footer">
            <time
            datetime="2026-02-28">2026-02-28</
            time>
            <span class="review-
            verified">✓ Verified purchase</span>
        </footer>
    </article>
    <article class="review-card">
        <header class="review-
        card-header">
            <span class="review-
            source">Walmart</span>
            <span class="review-
            author">J. Okafor</span>
            <span class="review-
            rating">5/5</span>
        </header>
        <h2>My go-to headphones</h2>
        <p>Comfortable, clear sound,
        pairs instantly with every device.</
        p>
        <footer class="review-
        card-footer">
            <time
            datetime="2026-04-02">2026-04-02</
            time>
            </footer>
        </article>
    <article class="review-card">
        <header class="review-
        card-header">
            <span class="review-
            source">Amazon</span>

```

```

        <span class="review-
author">L. Rossi</span>
        <span class="review-
rating">3/5</span>
    </header>
    <h2>Decent, not amazing</h2>
    <p>Audio is fine but the case
is flimsy and the mic is weak.</p>
    <footer class="review-
card-footer">

        <time
datetime="2026-01-19">2026-01-19</
time>

        <span class="review-
verified">✓ Verified purchase</span>
    </footer>
</article>
</section>
</main>

<footer>
    <p>&copy; 2026 Acme. All rights
reserved.</p>
</footer>
</body>
</html>

```

3 Part B — CSS: style and layout

CSS (Cascading Style Sheets) controls how the markup looks and where things sit. It is a list of **rules**, each a **selector** (which elements) plus a **declaration block** of `property: value` pairs. You keep it in `styles.css` and link it from the page, so one change restyles the whole review card.

```

/* selector { property: value; } */
.review-card {
    border: 1px solid #ddd;

```

```

padding: 1rem;
}

```

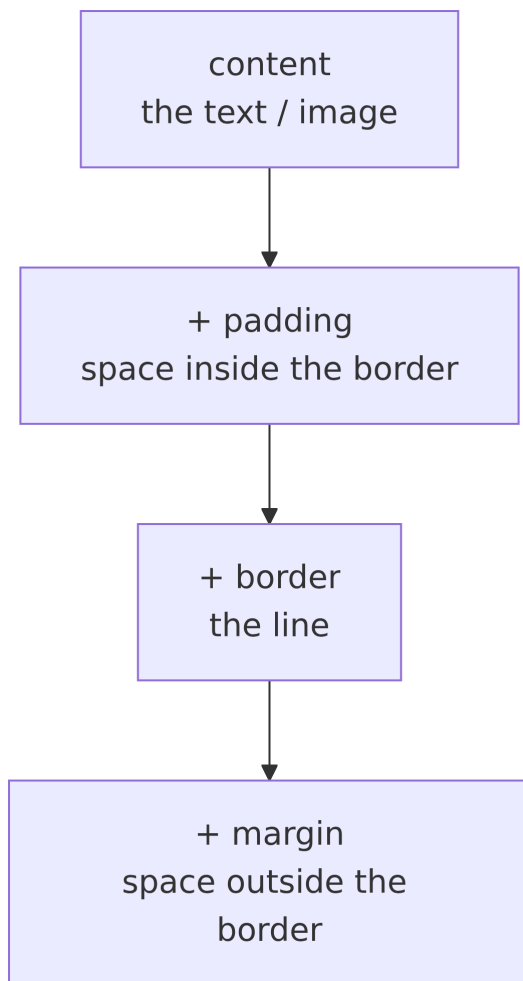
There are three ways to attach CSS to a page, in increasing order of specificity and decreasing order of reusability:

Way	Where	Use for
External	a <code>styles.css</code> file, via <code><link></code>	This course. Shared, cached, maintainable.
Internal	a <code><style></code> block in <code><head></code>	One-off page styles, or prototypes.
Inline	a <code>style="..."</code> attribute on an element	Quick overrides; avoid for real styling.

External wins because one file styles the page and is cached by the browser. It also sits at the *bottom* of the specificity ladder (section 12) — which is exactly why an inline `style="..."` can beat your `styles.css` rule when they disagree.

3.1 11. The box model

Every element is a **box** with four concentric layers, inside out: **content** → **padding** → **border** → **margin**.



- **content** — the size of the actual content (*width* / *height* refer to this).
- **padding** — space between the content and the border (inside the box).
- **border** — the line around the padding.
- **margin** — space between the box and its neighbours (outside the box). Margins of vertically-stacked boxes **collapse** to the larger of the two.

The single most important habit: set `box-sizing: border-box` globally so that *width* includes padding and border, not just the content. Without it, a “280px card” becomes 280px *plus* its padding and border, and your grid tracks stop lining up.

```
*, *::before, *::after {
  box-sizing: border-box;
}
```

A concrete walk-through: a card with *width*: 280px, *padding*: 1rem (16px each side), and a 1px border.

- With **border-box** (ours): the 280px *includes* the padding and border, so the content is $280 - 2 \times 16 - 2 \times 1 = \mathbf{246px}$ wide, and the box occupies exactly 280px.
- With **content-box** (the default): the 280px is the content *only*, so the box occupies $280 + 2 \times 16 + 2 \times 1 = \mathbf{314px}$ — and a grid track sized for 280px overflows.

That silent overflow is the bug **border-box** exists to prevent.

Worked example — a review card. The card is a box: the text content is the content, *padding* is the breathing room inside, *border* is the outline, and the *gap* between cards (set on the grid, section 15) plays the role of the space between them.

```
.review-card {
  border: 1px solid #ddd;
  border-radius: 8px;
  padding: 1rem;          /* inside the
border */
  background: #fff;
}
```

3.2 12. Selectors & specificity

A **selector** picks the elements a rule applies to. The ones you use every day:

Selector	Example	Picks
Element	<code>p</code>	Every <code><p></code> .
Class	<code>.review-card</code>	Every element with

Selector	Example	Picks
		<code>class="review-card"</code> .
ID	<code>#review-1</code>	The one element with <code>id="review-1"</code> .
Attribute	<code>input[type="email"]</code>	<code>email</code> inputs.
Descendant	<code>.review-card p</code>	Any <code><p></code> inside a review card.
Child	<code>.review-card > h2</code>	A <code><h2></code> that is a <i>direct</i> child of a review card.
Pseudo-class	<code>a:hover</code> , <code>a:focus</code>	A link in a given state.

Prefer classes for styling: they are reusable and carry moderate weight. Use element selectors for page-wide defaults (`body`, `p`, `h2`). **Avoid ID selectors for styling** — an ID is for anchors and for hooking up behaviour later; styling by ID is hard to override and couples your CSS to one specific element.

Specificity is the weight a selector carries when two rules target the same element and set the same property. It is scored weak → strong:

```
element < class / attribute /
pseudo-class < id < inline style
< !important
```

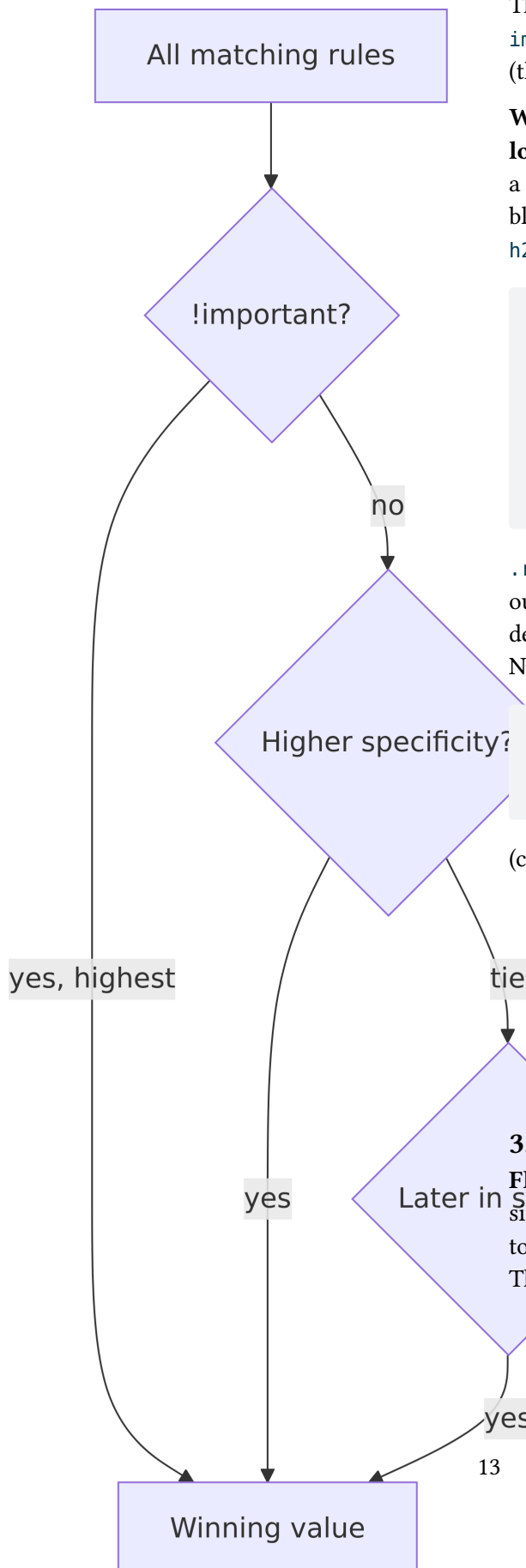
A class beats an element; an ID beats a class; an `inline style="..."` beats an ID; and `!important` beats everything — which is exactly why you should almost never need it.

Two more selector families you will meet:

- **Grouping** — `h1, h2, h3 { margin: 0 0 1rem; }` targets several selectors with one rule (comma-separated).
- **Pseudo-elements** — target a *part* of an element with a double colon: `::first-letter`, `::first-line`, `::before`, `::after`. The last two can generate content, for example a “Source:” prefix on every review’s source badge: `.review-card .review-source::before { content: "via "; }`.

3.3 13. The cascade — why one rule won

The **cascade** is the algorithm that decides which value wins when several rules apply to the same element and property. It resolves conflicts in a fixed order:



Three factors, in order: (1) **importance** (!important), (2) **specificity**, (3) **source order** (the later rule wins a tie).

Worked example — “why did my rule lose?” You want the review title to be dark, but a page-wide default is turning every heading blue. Two rules now fight over `.review-card h2`:

```

/* (a) page-wide default — element
specificity (0,0,1) */
h2 { color: #1e6fd6; }

/* (b) your review card rule — class
+ element specificity (0,1,1) */
.review-card h2 { color: #111; }

```

`.review-card h2` wins, because a **class** (0,1,1) outranks a bare **element** (0,0,1) — specificity decides before source order is even consulted. Now imagine you wrote the *opposite*:

```

/* (c) same element weight as (a),
but later in the file */
h2 { color: #111; }

```

(c) also beats (a) — same specificity, so **source order** breaks the tie and the later rule wins. The two lessons: raise specificity to *target* more precisely, and keep a predictable source order so ties resolve the way you expect. Reach for `!important` only to escape a third-party stylesheet, never inside your own code.

3.4 14. Layout with Flexbox

Flexbox lays out items along **one axis** — a single row or a single column. Use it for a toolbar, a nav bar, a card’s interior, or a header. The decision rule: **one axis** → **Flexbox**.

Set it on the **container**; the children become flex items. The properties you actually reach for:

Property (on container)	Does
<code>display: flex</code>	Makes the box a flex container.
<code>flex-direction</code>	<code>row</code> (default) or <code>column</code> — sets the main axis.
<code>justify-content</code>	Distributes items along the main axis (<code>space-between</code> , <code>center</code> , ...).
<code>align-items</code>	Aligns items along the cross axis (<code>center</code> , <code>stretch</code> , ...).
<code>gap</code>	Space between items (no more margin hacks).
<code>flex-wrap</code>	<code>wrap</code> lets items flow to a new line.

On an **item**, `flex: 1` lets it grow to fill the free space.

Worked example — the review card header. The header is a row: the source badge on the left, the author next, the rating pushed to the right. On a phone it stacks.

```
.review-card-header {
  display: flex;
  flex-direction: row; /*
  badge ... author ... rating */
  justify-content: space-between;
  align-items: center;
  gap: 0.5rem;
}
```

```
/* Mobile: stack badge above author
above rating */
@media (max-width: 600px) {
  .review-card-header { flex-
direction: column; align-items: flex-
start; }
}
```

```
<header class="review-card-header">
  <span class="review-
source">Amazon</span>
  <span class="review-author">M.
Dubois</span>
  <span class="review-rating">5/5</
span>
</header>
```

Read it as: **Flexbox positions one axis at a time**. The header is a `row`; on a phone it becomes a `column`.

The same tool, second example — the nav bar. A row of links that wraps onto a new line when the row fills up:

```
nav ul {
  display: flex;
  flex-wrap: wrap; /* links drop
to a new line when the row is full */
  gap: 1rem;
  list-style: none; /* keep the
list semantics; drop the bullets */
  padding: 0;
}
```

Flexbox keeps the semantic `<ul>` from section 2 but lays it out in a row — another reason that **semantics** (HTML) and **layout** (CSS) are separate concerns.

3.5 15. Layout with CSS Grid

Grid lays out items in **two dimensions** — rows and columns together. Use it for a card

gallery, a dashboard, or any “N things in a row that wrap”. The decision rule: **two axes** → **Grid**.

The key properties:

- `grid-template-columns` / `grid-template-rows` — define the tracks.
- `fr` — a fraction of the free space (`1fr` = one equal share).
- `gap` — spacing between tracks.
- `repeat(auto-fit, minmax(min, max))` — the responsive-card-grid workhorse.

A fixed grid, three equal columns:

```
.review-grid {
  display: grid;
  grid-template-columns: repeat(3,
1fr);
  gap: 1.5rem;
}
```

Worked example — the review grid as a responsive grid. The reviews page is a grid of review cards. With `minmax(280px, 1fr)` each track is “at least 280px, growing to fill its share”; with `auto-fit` the grid fits as many such tracks as the container allows. The column count now follows the viewport **with no media queries at all**:

```
.review-grid {
  display: grid;
  grid-template-columns: repeat(auto-
fit, minmax(280px, 1fr));
  gap: 1.5rem;
}
```

```
<section class="review-grid" aria-
label="All reviews">
  <article class="review-card">...</
article>
  <article class="review-card">...</
```

```
article>
  <article class="review-card">...</
article>
</section>
```

On a phone: one column. On a tablet: two or three. On a wide desktop: as many 280px+ tracks as fit. That is the whole idea.

You control rows the same way (`grid-template-rows`), but most card grids leave rows to `auto` — each row is as tall as its tallest card in that row. That is why a Grid of review cards plus a Flexbox card header (the source/author/rating row from section 14) gives you aligned rows *and* consistent card internals from two orthogonal layout decisions.

💡 Flexbox vs Grid, in one line

One axis → Flexbox; two axes → Grid.

In the review card, Grid lays out the cards on the page; Flexbox lays out the header and footer inside each card. Nesting Flexbox inside Grid cells is normal and expected.

3.6 16. Responsive design — media queries & fluid units, mobile-first

Mobile-first means you write the styles for the smallest screen first, then *enhance* upward. The base rules **are** the mobile rules; `min-width` media queries layer desktop behaviour on top as the viewport grows.

- **Media queries** gate a block of styles on a condition. `min-width` = mobile-first; `max-width` = desktop-first (start wide, shrink down — the pattern to avoid).
- **Breakpoints** are the widths you choose — pick them from where your *content* breaks,

not from habit. Common stops are around 600 / 768 / 1024 px.

- **Fluid units** keep things proportional instead of fixed:
 - `rem` — relative to the root font size; best for type and spacing.
 - `%` — relative to the parent; good for widths.
 - `vw` / `vh` — relative to the viewport.
 - `clamp(min, preferred, max)` — a self-limiting value, ideal for fluid type.

Worked example — the review grid, mobile-first. One column by default (mobile), then add columns as the viewport widens. (This is the explicit, breakpoint-driven alternative to the `auto-fit` trick in section 15 — both are valid; use whichever you can reason about.)

```
/* Base = mobile: one column */
.review-grid {
  display: grid;
  grid-template-columns: 1fr;
  gap: 1rem;
}
/* Tablet and up: two columns */
@media (min-width: 768px) {
  .review-grid { grid-template-columns: repeat(2, 1fr); gap: 1.5rem; }
}
/* Desktop and up: three columns */
@media (min-width: 1024px) {
  .review-grid { grid-template-columns: repeat(3, 1fr); }
}

/* Fluid type: scales between 1.25rem and 2rem */
.review-grid h2 {
  font-size: clamp(1.25rem, 1rem + 2vw, 2rem);
}
```

The `viewport` meta tag from section 1 is what makes all of this work on a real phone — without it, mobile browsers render at a virtual 980px-wide desktop and ignore your mobile base styles.

3.7 17. Tailwind CSS 3.4 — the app's real styling approach

Everything in Part B is **hand-written CSS** — and that is the point of this session: you must be able to read and reason about every rule before a framework writes it for you. But the app you are building the review layer for — `fullstack-minimal-app` — uses **Tailwind CSS 3.4**, a utility-first framework. You will meet it in every subsequent session, so here is the mapping.

The idea is the inverse of what you just learned: instead of naming a class and writing a rule in a stylesheet, you compose small single-purpose utilities straight in the markup. The concepts are identical — box model, Flexbox, Grid, responsive breakpoints — only the syntax moves into the HTML.

Hand-written CSS (what you built)	Tailwind 3.4 equivalent
<pre>.review-card { border: 1px solid #ddd; border-radius: 8px; padding: 1rem; }</pre>	<pre>class="border border-gray-300 rounded-lg p-4"</pre>
<pre>.review-card-header { display: flex; align-items: center; gap: 0.5rem; }</pre>	<pre>class="flex items-center gap-2"</pre>
<pre>.review-grid { display: grid; grid-template-columns: </pre>	<pre>class="grid grid-cols-3 gap-6"</pre>

Hand-written (what you built)	CSS	Tailwind 3.4 equivalent
	<pre>repeat(3, 1fr); gap: 1.5rem; }</pre>	
	<pre>@media (min-width: 768px) { .review-grid { grid-template-columns: repeat(2, 1fr); } }</pre>	<pre>class="grid grid-cols-1 md:grid-cols-2 gap-4"</pre>
	<pre>.review-source { background-color: #dbeafe; border-radius: 9999px; padding: 0.125rem 0.5rem; }</pre>	<pre>class="rounded-full bg-blue-100 text-blue-800 border-px-2 py-0.5"</pre>

Because you can already write the underlying CSS by hand, Tailwind will read as **shorthand rather than magic**. When you open the app's React components in later sessions, every utility class will map to a CSS rule you can reconstruct. That is the foundation this session lays: **the static CSS you wrote today is the vocabulary a utility framework speaks**.

4 Key takeaways

1. **One page, no scripting.** The whole review card is `index.html` + `styles.css`. Everything you learned maps to a rule you can read.
2. **Semantics first.** Landmarks, one `<h1>`, no skipped headings, `alt`, `time`, `article` — accessibility and SEO fall out of doing the HTML properly.
3. **box-sizing: border-box globally.** It is the difference between a grid that lines up and one that does not.

4. **Specificity, then source order.** A class beats an element; a tie goes to the later rule. That answers “why did my rule win / lose”.
5. **One axis → Flexbox; two axes → Grid.** Grid lays out the review cards; Flexbox lays out the header and footer inside each card.
6. **Mobile-first.** Base styles are the mobile styles; `min-width` queries and fluid units (`rem`, `%`, `clamp`) enhance upward.
7. **Tailwind 3.4 is the app's real styling approach.** The hand-written CSS you built today is the vocabulary Tailwind speaks — every utility maps to a rule you can reconstruct. In Session 3 the review card becomes a React component; the markup and layout you built here are the starting point, not the throwaway.