HTML SERIES · COURSE 1

HTML Fundamentals

Document structure, text, lists, links, media, tables, forms, semantic HTML, and a real project build. 12 complete chapters.

12 Chapters

osztromok.com

CHAPTER 1: WHAT HTML IS - DOCUMENT STRUCTURE

COURSE 1 · CH 1

What HTML Is — Document Structure

The doctype, the html/head/body skeleton, and what every single web page is actually built from

HTML (HyperText Markup Language) is not a programming language — it has no logic, no calculations, no conditionals. It's a **markup** language: plain text wrapped in tags that describe what each piece of content *is* (a heading, a paragraph, a link) rather than how it should look. That separation — structure in HTML, appearance in CSS, behaviour in JavaScript — is the foundation everything else in web development builds on.

The Minimal Valid HTML Document

```
<!DOCTYPE html>
<html>
<head>
  <title>My First Page</title>
</head>
<body>
  <h1>Hello, world!</h1>
</body>
</html>
```

Every single web page, no matter how complex, is built around this same skeleton. Breaking it down piece by piece:

`<!DOCTYPE html>`	Tells the browser "interpret this as modern HTML5" — without it, browsers fall back to older, inconsistent rendering quirks from decades ago. Must always be the very first line.
`<html>...</html>`	The root element — everything else in the document lives inside this single pair of tags.
`<head>...</head>`	Metadata about the page — title, character encoding, linked stylesheets/scripts. Nothing inside <head> is visible on the page itself.
`<title>...</title>`	The text shown in the browser tab and used as the default bookmark name — required inside <head>.
`<body>...</body>`	Everything the visitor actually sees and interacts with lives here — every heading, paragraph, image, button, and link.

HEAD VS BODY, IN ONE SENTENCE

If a piece of content is meant to be visible on the page, it goes in `<body>`. If it's information *about* the page itself — title, character set, links to CSS files — it goes in `<head>`.

Tags — Opening, Closing, and Self-Closing

Most HTML elements come in pairs: an opening tag and a matching closing tag, with content between them.

```
<p>This is a paragraph.</p>
```

Some elements never contain content and don't need a closing tag at all — these are **void elements**, and Chapter 5 (images) and Chapter 7 (forms) both use several of them.

```
<img src="photo.jpg" alt="A description">
<br>
<hr>
```

Attributes — Adding Information to a Tag

Attributes live inside the opening tag, as `name="value"` pairs, and add extra information or behaviour to an element. The `src` and `alt` in the `<img>` example above are both attributes.

id="..."

A unique identifier for one specific element — used by CSS and JavaScript to target it precisely. Must be unique within the whole page.

class="..."

A label that can be shared across many elements — the primary way CSS targets groups of similar elements (full coverage in the CSS courses).

lang="..."

Declares the document's (or an element's) language — affects screen readers, spell-check,

and search engines. Usually set once on <html> itself: <html lang="en">.

ALWAYS QUOTE ATTRIBUTE VALUES

`<img src=photo.jpg>` often works in practice, but it's technically invalid and breaks the moment the value contains a space or special character. `<img src="photo.jpg">` always works correctly — quoting consistently from the start avoids a class of bug that only shows up later.

Nesting and Indentation

Elements can contain other elements — and consistent indentation, while not required by the browser at all, makes a document's structure genuinely readable to a human looking at the source.

```
<body>
  <h1>Page Title</h1>
  <p>Some text with <strong>emphasis</strong> inside it.</p>
</body>
```

CHAPTER 1 QUICK REFERENCE

- **<!DOCTYPE html>** — always the first line, triggers modern rendering
- **<html>** — the root element containing everything else
- **<head>** — metadata, not visible on the page; **<body>** — everything visible
- **<title>** — required inside <head>, shown in the browser tab
- **Opening/closing tag pairs** wrap content; **void elements** (img, br, hr) never do
- **Attributes** — name="value" pairs inside the opening tag; always quote the value
- **id** — unique per page; **class** — shareable across many elements
- **Next chapter:** text basics — headings, paragraphs, and text formatting tags

CHAPTER 2: TEXT BASICS

COURSE 1 · CH 2

Text Basics

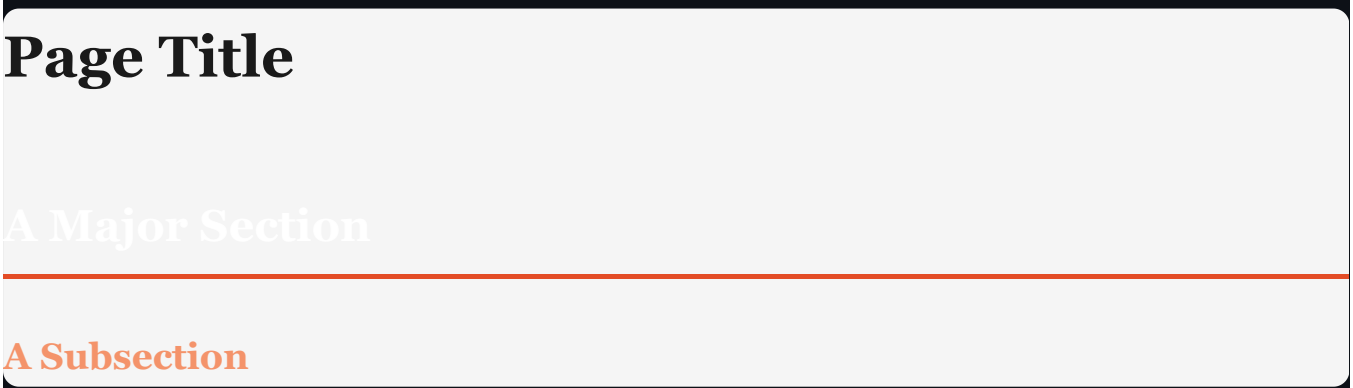
Headings, paragraphs, and the text formatting tags that give written content meaning, not just emphasis

Almost every web page is, at its core, structured text — and HTML provides a specific set of tags for exactly that. The key idea running through this entire chapter: HTML tags describe *what* a piece of text means semantically, while CSS (covered in the separate CSS courses) controls how it actually looks. Choosing the right tag matters even before any styling is applied.

Headings — h1 Through h6

Six levels of heading, from most important (`<h1>`) to least (`<h6>`) — used to create a logical outline of the page's content, similar to chapter and section headings in a book.

```
<h1>Page Title</h1>
<h2>A Major Section</h2>
<h3>A Subsection</h3>
```



NEVER SKIP HEADING LEVELS JUST TO GET A SMALLER DEFAULT SIZE

Using `<h4>` instead of `<h2>` purely because it renders smaller breaks the document's logical outline — screen readers and search engines both rely on heading levels reflecting genuine document structure. If a heading needs to look smaller, use CSS to style an `<h2>` rather than reaching for a lower heading number for purely visual reasons.

Paragraphs

```
<p>This is one paragraph of text. It can span multiple
sentences and even multiple lines in the source code — the
browser doesn't care about line breaks inside the tag itself.</p>
<p>This is a separate paragraph.</p>
```

Browsers automatically add space between separate `<p>` elements — there's no need to manually insert `<br>` tags between paragraphs to create that visual gap.

Text Formatting Tags

`<strong>`

Marks text as having strong importance — typically rendered bold. Semantic: "this matters."

`<em>`

Marks text with emphasis — typically rendered italic. Semantic: "say this differently when read aloud."

`<b>`

Bold text with NO implied importance — purely visual. Rare to need this over in practice.

`<i>`

Italic text with NO implied emphasis — used for things like foreign words or technical terms, not for stressing a point.

`<small>`

Side comments, fine print, legal disclaimers — semantically "less important," not just visually smaller.

`<mark>`

Highlighted text, as if marked with a highlighter pen — often used to show search-match results.

`<sub>` / `<sup>`

Subscript and superscript — chemical formulas (H_2O), footnote markers, mathematical notation.

`<br>`

A single line break WITHIN a block of text — a void element (Chapter 1). Should be rare; most spacing should come from proper paragraph/heading structure, not manual line breaks.

STRONG/EM VS B/I — THE DISTINCTION IS MEANING, NOT APPEARANCE

Visually, `<strong>` and `<b>` often look identical (bold), and the same goes for `<em>` and `<i>` (italic). The real difference is semantic: screen readers may change their tone of voice for `<strong>` / `<em>` content, and search engines weigh them differently — `<b>` / `<i>` carry no such meaning, purely visual styling with zero semantic weight.

Horizontal Rules and Line Breaks

```
<p>End of one section.</p>
<hr>
<p>Start of the next, visually separated section.</p>
```

`<hr>` represents a thematic break between sections of content — not just a decorative horizontal line, even though that's how it renders by default.

A Realistic Example, Combined

```
<h1>Product Review</h1>
<p>This product is <strong>excellent value</strong> for the price,
though I <em>really</em> wish the battery lasted longer.</p>
<p><small>This review reflects my personal experience only.</small></p>
```

CHAPTER 2 QUICK REFERENCE

- **h1-h6** — logical document outline, never skip levels for visual sizing alone
- **p** — paragraphs, with automatic spacing between them
- **strong/em** — semantic importance/emphasis; **b/i** — purely visual, no semantic meaning
- **small** — fine print/disclaimers; **mark** — highlighted text
- **sub/sup** — subscript/superscript
- **br** — a single line break within text, used sparingly
- **hr** — a thematic break between sections, not just a decorative line
- **Next chapter:** lists — ordered, unordered, and definition lists

CHAPTER 3: LISTS

COURSE 1 · CH 3

Lists

Ordered lists, unordered lists, and definition lists — and choosing the right one for the content

HTML provides three distinct list types, and each one carries a specific meaning — using the right one matters for the same semantic reasons covered in Chapter 2's heading and text-formatting discussion.

Unordered Lists —

For items where order doesn't matter — a list of features, ingredients, or general bullet points.

```
<ul>
  <li>HTML</li>
  <li>CSS</li>
  <li>JavaScript</li>
</ul>
```

- HTML
- CSS
- JavaScript

Ordered Lists —

For items where sequence genuinely matters — steps in a recipe, ranked results, instructions that must happen in order.

```
<ol>
  <li>Preheat the oven</li>
  <li>Mix the ingredients</li>
```

```
  <li>Bake for 20 minutes</li>
</ol>
```

1. Preheat the oven
2. Mix the ingredients
3. Bake for 20 minutes

CHOOSING UL VS OL ISN'T ABOUT HOW IT SHOULD LOOK — CSS CAN RESTYLE EITHER ONE

A common beginner mistake is picking `<ol>` because numbers are wanted visually, or `<ul>` because bullets are wanted — but CSS can make an ordered list show bullets, or an unordered list show numbers, entirely independent of the tag used. The choice should be based purely on whether sequence is semantically meaningful to the content, not on the default visual style.

Useful <ol> attributes

```
<ol start="5"> ...starts counting from 5 instead of 1
<ol reversed> ...counts downward instead of up
```

Nested Lists

A `<ul>` or `<ol>` can be nested inside an `<li>` of another list, to any depth — useful for sub-steps or hierarchical groupings.

```
<ul>
  <li>Frontend
    <ul>
      <li>HTML</li>
      <li>CSS</li>
    </ul>
  </li>
  <li>Backend</li>
</ul>
```

Definition Lists — <dl>

The least commonly used of the three, but exactly right for term/definition pairs — a glossary, a list of metadata key-value pairs, FAQ-style content.

```
<dl>
  <dt>HTML</dt>
  <dd>HyperText Markup Language — the structure of a web page.</dd>
  <dt>CSS</dt>
  <dd>Cascading Style Sheets — the visual presentation of a page.</dd>
</dl>
```

HTML
: HyperText Markup Language — the structure of a web page.

CSS
: Cascading Style Sheets — the visual presentation of a page.

<dl>

The wrapping element — "definition list."

<dt>

"Definition term" — the thing being defined.

<dd>

"Definition description" — the explanation. Multiple <dd> per <dt> are allowed, for several definitions of the same term.

Choosing the Right List Type

- **Sequence matters?** Use `<ol>` — steps, rankings, instructions.
- **Sequence doesn't matter?** Use `<ul>` — features, general groupings, navigation menus (a very common real-world use, often styled to look nothing like a bulleted list via CSS).
- **Term/explanation pairs?** Use `<dl>` — glossaries, metadata, FAQs.

NAVIGATION MENUS ARE ALMOST ALWAYS BUILT FROM <UL> UNDER THE HOOD

A site's main navigation bar — even one that visually looks like a row of buttons with no bullets at all — is conventionally marked up as a `<ul>` of links, then heavily restyled with CSS. The semantic

meaning ("this is a list of navigation options") stays intact for screen readers and search engines, even though the bullets and vertical stacking are completely styled away.

CHAPTER 3 QUICK REFERENCE

- **<ul>** — unordered, sequence doesn't matter; **<ol>** — ordered, sequence matters
- **<li>** — each list item, used inside either <ul> or
- **start / reversed** attributes on <ol> control numbering behaviour
- **Choosing ul vs ol** is about semantic sequence meaning, not visual bullets vs numbers — CSS can restyle either
- **Lists nest** inside <li> elements to any depth
- **<dl>/<dt>/<dd>** — definition lists for term/explanation pairs (glossaries, FAQs, metadata)
- **Navigation menus** are conventionally a <ul> of links, however differently they end up styled
- **Next chapter:** links & navigation — the anchor tag, attributes, relative vs absolute URLs

CHAPTER 4: LINKS & NAVIGATION

COURSE 1 · CH 4

Links & Navigation

The anchor tag, its key attributes, and the crucial difference between relative and absolute URLs

Links are what make the web a *web* — without them, every page would be an isolated island. The anchor tag, `<a>`, is how HTML creates a link, and understanding exactly how it resolves a URL is one of the most practically important things in this entire course — getting it wrong is one of the most common real-world bugs in deployed websites.

The Anchor Tag

```
<a href="https://example.com">Visit Example</a>
```

`href` ("hypertext reference") is the one required attribute — it specifies the destination. Without it, an `<a>` tag isn't actually a link at all, just plain inline text.

Absolute vs Relative URLs

This is the single most consequential distinction in this chapter — and the difference between a link that works correctly everywhere and one that quietly breaks the moment a site moves or restructures.

🌐 Absolute URL

The complete address, including the protocol and domain: `https://osztromok.com/linux`

Always points to the exact same place, regardless of where the link itself lives — but means the link breaks if the target domain ever changes.

📁 Relative URL

A path relative to the *current page's* own location: `/linux` or `../images/photo.jpg`

Continues working correctly even if the whole site moves to a new domain — exactly why internal links should almost always be relative.

```
// Absolute — full URL, works regardless of context, but hardcodes the domain
<a href="https://osztromok.com/linux">Linux course</a>

// Relative to the site root — starts with /
<a href="/linux">Linux course</a>

// Relative to the CURRENT page's folder — no leading slash
<a href="chapter2.html">Next chapter</a>

// Relative, going UP one folder level first — ../
<a href="../images/photo.jpg">View photo</a>
```

USE ABSOLUTE URLS ONLY FOR GENUINELY EXTERNAL LINKS

Hardcoding `https://osztromok.com/...` for links *within* osztromok.com itself works, but breaks immediately if the domain or protocol ever changes (exactly the kind of migration covered in the website rebuild courses). Internal links should be relative; absolute URLs are for linking to genuinely different domains.

Linking Within a Page — Fragment Identifiers

A `#` followed by an element's `id` (Chapter 1) jumps to that specific element on the page — the basis of "back to top" links and in-page tables of contents.

```
<a href="#section2">Jump to Section 2</a>

// ...further down the same page:
<h2 id="section2">Section 2</h2>
```

This same syntax can also follow a full URL to jump directly to a specific section of *another* page: `https://osztromok.com/linux#chapter3`.

Key Anchor Attributes

ATTRIBUTE	WHAT IT DOES
`target="_blank"`	Opens the link in a new tab/window instead of navigating away from the current page

ATTRIBUTE	WHAT IT DOES
`rel="noopener"`	A security measure, used alongside target="_blank" — see warning below
`download`	Tells the browser to download the linked file rather than navigate to/display it
`title`	Tooltip text shown on hover — supplementary, not a replacement for clear link text itself

```
<a href="https://example.com" target="_blank" rel="noopener">External site</a>
```

ALWAYS PAIR TARGET="_BLANK" WITH REL="NOOPENER"

Without it, the newly opened page gains a (limited but real) ability to access and manipulate the original page via JavaScript's `window.opener` — a known security/performance concern. `rel="noopener"` severs that connection, and modern best practice treats it as required any time `target="_blank"` is used.

Writing Good Link Text

- **Avoid "click here" as link text.** Screen reader users often navigate by jumping between links in a list — "click here" repeated multiple times on a page gives zero useful context out of that surrounding context.
- **Describe the destination.** "Read the full Linux installation guide" tells a user (and a search engine) far more than "click here" or "read more."
- **Don't use a raw URL as the visible text** unless the URL itself is genuinely the point — it's rarely readable or memorable as link text.

CHAPTER 4 QUICK REFERENCE

- **<a href="...">** — href is the one required attribute; without it, not actually a link
- **Absolute URL** — full address with protocol/domain; use for genuinely external links
- **Relative URL** — resolved against the current page's location; use for internal links so they survive domain/protocol changes
- **#id** — jumps to a specific element on the current (or another) page
- **target="_blank"** — opens in a new tab; always pair with **rel="noopener"**
- **Write descriptive link text** — never "click here," describe the actual destination

- **Next chapter:** images & media — img, alt text, figure/figcaption, basic audio/video

CHAPTER 5: IMAGES & MEDIA

COURSE 1 · CH 5

Images & Media

img and alt text done properly, figure/figcaption, and the basics of embedding audio and video

Images are where this course's recurring theme — semantic meaning, not just visual appearance — matters most concretely. The single most important attribute covered in this chapter, `alt`, has real consequences for accessibility, SEO, and what happens when an image simply fails to load.

The <img> Tag

```
<img src="sunset.jpg" alt="Orange sunset over a calm ocean">
```

A void element (Chapter 1) — no closing tag. `src` is the image's location (relative or absolute, exactly as covered for links in Chapter 4); `alt` is a text description of the image's content.

Why alt Text Actually Matters

- **Screen readers read it aloud** — for a visually impaired user, alt text IS the image, the only way they experience it at all.
- **It displays if the image fails to load** — a broken image link, a slow connection, or a typo'd filename all show the alt text as a fallback instead of nothing.
- **Search engines use it** — image search results and general SEO both weigh alt text meaningfully.

Writing genuinely good alt text

POOR

alt="image123.jpg"

BETTER

alt="A red bicycle leaning against a brick wall"

POOR (REDUNDANT)

alt="Picture of a sunset"

BETTER (NO NEED TO SAY "PICTURE OF")

alt="Orange sunset over a calm ocean"

A PURELY DECORATIVE IMAGE SHOULD HAVE AN EMPTY ALT, NEVER A MISSING ONE

For an image that adds nothing semantically (a decorative divider graphic, a background flourish), use `alt=""` — an explicit empty string. This tells screen readers to skip it silently, rather than reading out a filename or, worse, having no `alt` attribute at all, which is technically invalid HTML and produces inconsistent, often worse, fallback behaviour across different browsers and screen readers.

Sizing Attributes

```
<img src="photo.jpg" alt="..." width="400" height="300">
```

Specifying both `width` and `height` (in pixels, no unit needed in the attribute itself) lets the browser reserve the correct space for the image *before* it finishes loading — preventing the rest of the page's content from visibly jumping around as images load in, a real and measurable user-experience problem on slower connections.

figure and figcaption — Images with Captions

When an image needs a caption, `<figure>` and `<figcaption>` together provide the correct semantic wrapper — rather than just placing a `<p>` next to an `<img>` with no formal relationship between them.

```
<figure>
  <img src="chart.png" alt="Bar chart showing quarterly sales growth">
  <figcaption>Figure 1: Sales grew 23% in Q3 2026</figcaption>
</figure>
```

Basic Audio

```
<audio controls>
  <source src="podcast.mp3" type="audio/mpeg">
  Your browser doesn't support the audio element.
</audio>
```

The `controls` attribute shows the browser's built-in play/pause/volume UI — without it, audio loads but gives the visitor no way to interact with it at all. The text inside the tags is a fallback shown only on genuinely ancient browsers that don't support `<audio>`.

Basic Video

```
<video controls width="640">
  <source src="demo.mp4" type="video/mp4">
  Your browser doesn't support the video element.
</video>
```

ATTRIBUTE	WHAT IT DOES
controls	Shows the built-in playback UI — almost always wanted
autoplay	Starts playing immediately on page load — use very sparingly; most browsers mute autoplay by default unless the muted attribute is also present
loop	Restarts automatically when playback ends
muted	Starts with no sound — often required alongside autoplay for it to work at all
poster	(video only) An image shown before playback starts, instead of a blank/black frame

AVOID AUTOPLAY WITH SOUND — IT'S A NEAR-UNIVERSALLY DISLIKED PATTERN

Most browsers actively block autoplaying audio with sound by default specifically because of how disruptive and unwanted it is for visitors. If autoplay is genuinely needed for a background visual effect, pair it with `muted` — that combination is reliably allowed across browsers; autoplay with sound, generally, is not.

CHAPTER 5 QUICK REFERENCE

- **<img src="..." alt="...">** — alt is read by screen readers, shown if the image fails, and used by search engines
- **Decorative images:** alt="" (explicit empty), never a missing alt attribute entirely
- **width/height attributes** reserve layout space before the image loads, preventing visible content jumping
- **<figure>/<figcaption>** — the correct semantic pairing for a captioned image

- **<audio controls>/<video controls>** — controls attribute is almost always wanted
- **autoplay + muted** is reliably allowed; autoplay with sound generally is not, and is widely disliked anyway
- **Next chapter:** tables — structure, headers, and basic accessibility

CHAPTER 6: TABLES

COURSE 1 · CH 6

Tables

Structure, headers, and the basic accessibility considerations that make a table actually usable, not just visually gridded

HTML tables exist for one specific purpose: presenting genuinely **tabular data** — rows and columns of related values, like a spreadsheet. They were widely misused in the early web era for general page layout; this course treats that as firmly settled history — modern layout belongs entirely to CSS, covered in the separate CSS courses, and tables here are strictly for actual tabular data.

Basic Table Structure

```
<table>
  <tr>
    <th>Name</th>
    <th>Role</th>
  </tr>
  <tr>
    <td>Philip</td>
    <td>Admin</td>
  </tr>
</table>
```

Name	Role
Philip	Admin

<table>	<tr>	<th>	<td>
The whole table	Table row	Header cell	Data cell

thead, tbody, tfoot — Grouping Rows Semantically

For anything beyond a trivial table, explicitly grouping the header row(s), body rows, and any footer row improves both semantics and styling hooks for CSS.

```
<table>
  <thead>
    <tr>
      <th>Product</th>
      <th>Price</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Widget</td>
      <td>£9.99</td>
    </tr>
  </tbody>
  <tfoot>
    <tr>
      <td>Total</td>
      <td>£9.99</td>
    </tr>
  </tfoot>
</table>
```

caption — Describing the Table's Purpose

```
<table>
  <caption>Q3 2026 Regional Sales</caption>
  // ...rows...
</table>
```

A genuine accessibility feature, not just a visual label — screen readers announce the caption before reading the table's content, giving context about what the data actually represents before diving into individual cells.

scope — The Single Most Important Accessibility Attribute for Tables

```
<th scope="col">Name</th>
<th scope="row">Philip</th>
```

`scope="col"` tells assistive technology this header applies to the entire column beneath it; `scope="row"` means it applies to the entire row alongside it. Without explicit scope, a screen reader has to guess at the relationship between headers and data cells — for anything beyond the simplest table, this can produce genuinely confusing or incorrect announcements.

A VISUALLY-STYLED TABLE THAT USES DIVS INSTEAD OF REAL TABLE TAGS LOSES ALL OF THIS FOR FREE

It's tempting, especially with modern CSS layout tools, to build something that "looks like a table" out of styled `<div>`s — but doing so discards every piece of built-in table semantics covered in this chapter: row/column relationships, scope announcements, and caption support all rely on the browser recognising genuine table markup. If the data is actually tabular, use real table tags.

Spanning Cells — colspan and rowspan

```
<tr>
  <th colspan="2">Combined Header Spanning 2 Columns</th>
</tr>
```

`colspan` makes a cell span multiple columns; `rowspan` makes a cell span multiple rows. Genuinely useful for grouped headers, but worth using sparingly — heavily merged tables can become harder, not easier, for both sighted users and screen readers to follow correctly.

CHAPTER 6 QUICK REFERENCE

- **table/tr/th/td** — table, row, header cell, data cell — the four core building blocks
- **thead/tbody/tfoot** — semantic row grouping, used for anything beyond a trivial table
- **caption** — announced by screen readers before the table's content, real accessibility value
- **scope="col" / scope="row"** — the single most important table accessibility attribute, clarifies header-to-cell relationships
- **colspan/rowspan** — merge cells across columns/rows; use sparingly
- **Tables are for tabular data only** — never for general page layout, which belongs to CSS

- **Next chapter:** forms basics — input types, labels, and basic form structure

CHAPTER 7: FORMS BASICS

COURSE 1 · CH 7

Forms Basics

Input types, labels, and the basic structure every HTML form is built from

Forms are how a web page collects information from a visitor — search boxes, login screens, contact forms, checkout flows. This chapter covers the foundational structure; Intermediate Course Chapter 1 goes much deeper into validation, fieldsets, and the full range of input types.

The Basic Form Structure

```
<form action="/submit" method="post">
  <label for="email">Email address</label>
  <input type="email" id="email" name="email">

  <button type="submit">Subscribe</button>
</form>
```

- **action** — where the form's data gets sent when submitted.
- **method** — how it's sent: `get` appends data to the URL (visible, bookmarkable, size-limited); `post` sends it in the request body (not visible in the URL, no practical size limit).

NEVER USE METHOD="GET" FOR ANYTHING SENSITIVE

A password or any private data submitted via GET ends up directly in the URL — visible in browser history, server logs, and potentially shared accidentally if someone copies the URL. Always use POST for login forms, anything containing personal information, or anything that changes data on the server.

label and for — The Most Important Form Accessibility Pairing

The `for` attribute on a `<label>` must exactly match the `id` of its corresponding input — this creates a genuine, browser-recognised association, not just visual proximity.

```
<label for="username">Username</label>
```

```
<input type="text" id="username" name="username">
```

Why this matters beyond just looking right

- **Screen readers** announce the label when the input receives focus — without the for/id link, a sighted user sees the label, but it may be completely disconnected for a screen reader user.
- **Clicking the label text itself** focuses or activates the associated input — genuinely useful on mobile, where tapping a small checkbox/radio button precisely is harder than tapping its larger text label.

> **WRAPPING THE INPUT INSIDE THE LABEL ACHIEVES THE SAME ASSOCIATION WITHOUT NEEDING MATCHING IDS**
>
> `<label>Username <input type="text" name="username"></label>` creates the same accessible association implicitly, through nesting rather than matching id/for values — a valid alternative, though the explicit for/id pairing is generally considered clearer to read in larger forms.

Common Input Types

`text` Single-line free text	**`email`** Text, with basic format validation and a relevant mobile keyboard	**`password`** Text masked with dots/asterisks as it's typed
`number` Numeric input, often with up/down spinner controls	**`checkbox`** A single on/off toggle	**`radio`** One choice from a group sharing the same name
`submit` A button that submits the form	**`checkbox/radio note`** Full coverage of all remaining types in Intermediate Ch1	

name — The Attribute That Actually Submits Data

Easy to overlook: an input needs a `name` attribute for its value to actually be included when the form is submitted — `id` alone (used for the label association above) is not enough on its own.

```
// This input's value is NEVER submitted — no name attribute
<input type="text" id="username">

// This one IS submitted, as "username=whatever-was-typed"
<input type="text" id="username" name="username">
```

Radio Buttons — Grouped by a Shared name

```
<input type="radio" id="small" name="size" value="small">
<label for="small">Small</label>

<input type="radio" id="large" name="size" value="large">
<label for="large">Large</label>
```

Both radio buttons share `name="size"` — that shared name is what makes them mutually exclusive (selecting one deselects the other); each still needs its own unique `id` for its label to work correctly.

CHAPTER 7 QUICK REFERENCE

- **<form action="..." method="..."> ** — where data goes, and how it's sent (GET vs POST)
- **Never use GET for sensitive data** — it ends up visible in the URL
- **label for="x" + input id="x"** — the core accessibility pairing; alternatively, nest the input inside the label
- **name attribute** — required for an input's value to actually be submitted; id alone isn't enough
- **Radio buttons** — grouped into a mutually exclusive set via a shared name, with unique ids per option
- **Common types:** text, email, password, number, checkbox, radio, submit — full range covered in Intermediate Ch1
- **Next chapter:** semantic HTML — header, nav, main, article, section, footer

CHAPTER 8: SEMANTIC HTML

COURSE 1 · CH 8

Semantic HTML

header, nav, main, article, section, footer — describing a page's structure meaningfully, not just visually

Every chapter so far has touched on semantic meaning for specific pieces of content — headings, text emphasis, lists, tables. This chapter applies the same principle at the level of overall page layout: describing *what role* each major region of a page plays, using purpose-built tags instead of generic containers.

A Typical Page, Structured Semantically

`<header>`
Site logo, title, primary navigation — the top of the page

- `<nav>`
  The main navigation menu, often inside header

`<main>`
The primary, unique content of THIS page — exactly one per page

- `<article>`
  A self-contained piece of content — a blog post, a news story
- `<section>`
  A thematic grouping within the content, usually with its own heading

`<footer>`
Copyright, contact info, secondary links — the bottom of the page

The Elements, One at a Time

`<header>`

Introductory content for the page (or for a section/article within it — header isn't exclusively a page-top element). Often contains the logo and main navigation.

`<nav>`

A block of navigation links — the main menu, a table of contents, breadcrumbs. Not every group of links needs <nav>, just the genuinely major navigation blocks.

<main>

The dominant, unique content of the page — excludes repeated elements like the header, footer, and sidebar navigation. Exactly one <main> per page.

<article>

Content that would make sense distributed/syndicated on its own — a blog post, a forum post, a news article. The test: "could this stand alone elsewhere and still make sense?"

<section>

A thematic grouping of content, typically with its own heading — less independent than <article>, more meaningful than a generic .

<aside>

Content tangentially related to the main content — a sidebar, a pull quote, related links — not essential to understanding the primary content.

<footer>

Closing content — copyright, contact details, sitemap links. Like header, can apply to the whole page or to an individual section/article.

article vs section — The Distinction That Actually Trips People Up

This is the single most commonly confused pair in semantic HTML. The practical test: **would this content still make complete sense if it were pulled out and placed entirely on its own, somewhere else?**

```
// A blog post — genuinely independent, could be syndicated via RSS as-is
<article>
  <h2>How to Set Up a Raspberry Pi</h2>
  <p>...</p>
</article>

// A themed grouping WITHIN that article — not independent on its own
<article>
  <h2>How to Set Up a Raspberry Pi</h2>
  <section>
    <h3>Required Hardware</h3>
    <p>...</p>
  </section>
</article>
```

IT'S GENUINELY FINE IF A PAGE ONLY USES A DIV IN SOME CASES

Not every grouping needs a semantic tag — a `<section>` without its own heading, used purely as a styling hook, is arguably better written as a plain `<div>`. The semantic elements in this chapter exist for cases where the grouping carries real, document-outline-relevant meaning — forcing semantic tags onto every container for their own sake misses the actual point.

Why This Matters Beyond Just Convention

- **Screen readers** let users jump directly between landmarks (header, nav, main, footer) — genuine, measurable navigation efficiency for assistive technology users.
- **Search engines** use this structure to better understand which content on a page is the primary content versus boilerplate navigation/footer text.
- **Other developers** (including future you) read semantically structured markup faster than a page built entirely from generically named divs.

"DIV SOUP" — AN ENTIRE PAGE BUILT FROM NESTED, UNLABELLED DIVS — IS THE THING THIS CHAPTER EXISTS TO PREVENT

A page where every region is `<div class="top">`, `<div class="content">`, `<div class="bottom">` works visually but conveys zero structural meaning to anything other than a sighted human reading the rendered page — exactly the gap semantic HTML closes.

CHAPTER 8 QUICK REFERENCE

- **header / nav / main / footer** — top-level page landmarks; exactly one <main> per page
- **article** — independently meaningful content (could stand alone elsewhere)
- **section** — thematic grouping, usually with its own heading, less independent than article
- **aside** — tangentially related content, not essential to the main content
- **article vs section test:** "would this make sense entirely on its own, elsewhere?"
- **Not every container needs a semantic tag** — a plain div is fine for purely visual/styling groupings
- **Why it matters:** screen reader landmark navigation, SEO content understanding, and human readability of the source
- **Next chapter:** divs & spans — generic containers, and when to use them vs semantic tags

CHAPTER 9: DIVS & SPANS

COURSE 1 · CH 9

Divs & Spans

Generic containers with no inherent meaning — and exactly when reaching for them is the right call, not a shortcut

Chapter 8 covered semantic elements that carry real structural meaning. This chapter covers their deliberate opposite: `<div>` and `<span>`, which carry **zero** semantic meaning by design — and that's precisely what makes them useful for the cases semantic tags don't cover.

div vs span — The One Real Difference

A generic **block-level** container — takes up its own line, full available width by default. Used to group larger chunks of content for styling or scripting purposes.

A generic **inline** container — flows within surrounding text, doesn't force a line break. Used to wrap a small piece of text or inline content for styling purposes.

```
// div — wraps a whole block of content
<div class="card">
  <h3>Card Title</h3>
  <p>Card content goes here.</p>
</div>

// span — wraps a piece of text WITHIN a sentence
<p>The total is <span class="price">£42.00</span>, including tax.</p>
```

When a div or span Is Genuinely the Right Choice

- **Purely a styling/layout hook with no semantic meaning of its own.** A wrapper used to apply a CSS grid or flexbox layout to several unrelated elements together has no natural semantic tag — div is correct here, not a workaround.

- **Wrapping content for JavaScript to target.** A container that needs a hook for show/hide logic, with no inherent structural meaning, is exactly what div/span exist for.
- **A small inline piece of text needing its own style** — highlighting a specific word or value within a sentence, where no existing semantic inline tag (Chapter 2's strong/em/mark, etc.) actually fits the meaning.

When to Reach for a Semantic Tag Instead

INSTEAD OF...	USE	BECAUSE
`<div class="header">`	`<header>`	Genuine page/section landmark meaning (Chapter 8)
`<div class="nav">`	`<nav>`	Navigation block, recognised by screen readers
`<span class="bold">`	`<strong>`	Genuine importance, not just visual boldness (Chapter 2)
`<div class="list-item">`	`<li>` inside `<ul>/<ol>`	Genuine list semantics (Chapter 3)
`<div class="link">` with onclick	`<a href="...">`	Real navigation, keyboard-accessible by default (Chapter 4)

THIS ENTIRE PATTERN OF MISUSE HAS A NAME — "DIV SOUP" OR "DIVITIS"

Reaching for div/span as a default first instinct, rather than checking whether a semantic tag already fits, is the single most common HTML anti-pattern. Chapter 8's "div soup" warning and this chapter's misuse table describe the same underlying problem from two directions — every semantic tag covered across this course exists specifically to handle a case that a generic div/span would otherwise be misused for.

A Practical Test Before Reaching for div/span

1. Does this content's purpose match an existing semantic tag from Chapters 1–8 (header, nav, main, article, section, strong, em, etc.)? If yes, use that.
2. Is this purely a styling/scripting container with genuinely no semantic role of its own? If yes, div/span is correct, not a shortcut.
3. If genuinely unsure, lean toward the semantic option — it costs nothing extra and provides real value to accessibility and SEO that a div never will.

DIV AND SPAN ARE NOT "BAD" — THEY'RE JUST NOT THE ANSWER TO EVERYTHING

A complex, real website typically uses div/span constantly, often dozens of times per page — for layout wrappers, styling hooks, and grouping that has no semantic meaning of its own. The goal of this chapter isn't avoiding them entirely; it's making sure they're a deliberate choice for genuinely non-semantic content, not a reflexive default used everywhere a more meaningful tag would have fit just as easily.

CHAPTER 9 QUICK REFERENCE

- ** ** — generic block-level container; **** — generic inline container
- **Both carry zero semantic meaning** by design — that's the entire point of their existence
- **Correct use:** pure styling/layout wrappers, JS scripting hooks, with no existing semantic tag that fits better
- **Before using div/span, check** whether a semantic tag from Chapters 1-8 already fits the content's actual purpose
- **"div soup"/"divitis"** — the anti-pattern of using generic containers everywhere by reflex, ignoring better-fitting semantic options
- **div/span aren't "bad"** — they're genuinely necessary, just not a universal default
- **Next chapter:** comments, whitespace, and validation — writing clean, valid HTML

CHAPTER 10: COMMENTS, WHITESPACE, VALIDATION

COURSE 1 · CH 10

Comments, Whitespace, and Validation

Writing HTML that's clean, well-documented where it needs to be, and genuinely valid — not just "happens to render correctly"

Browsers are remarkably forgiving of invalid HTML — they'll often render something reasonable-looking even when the underlying markup is technically broken. That forgiveness is also exactly why bugs in HTML structure can go unnoticed for a long time. This chapter covers comments, whitespace conventions, and how to actually verify a document is valid rather than just "looks fine in one browser."

HTML Comments

```
<!-- This is a comment — never shown to visitors, only visible in the source -->
<p>This text IS visible.</p>

<!-- TODO: replace this placeholder image before launch -->
<img src="placeholder.jpg" alt="Placeholder">
```

Useful for:

- **Marking section boundaries** in a long file — `<!-- End of header -->` after a long closing block.
- **Leaving notes for future maintainers** (including future you) about why something is structured a particular way.
- **Temporarily disabling a block** during development, without deleting it outright.

COMMENTS ARE VISIBLE TO ANYONE WHO VIEWS THE PAGE SOURCE

Unlike a comment in a compiled programming language, an HTML comment is sent to the browser exactly as written and visible to anyone who opens DevTools or "View Source." Never leave sensitive information, internal notes about security weaknesses, or anything not meant for public eyes in an HTML comment.

Whitespace — Why It (Mostly) Doesn't Matter, and When It Does

Browsers collapse multiple consecutive whitespace characters (spaces, tabs, newlines) into a single space when rendering text content — this is why indentation and line breaks in the source code are purely for human readability, not something the browser preserves visually by default.

```
<p>This    has     extra     spaces
    and a line break
    in the source.</p>
// Renders as: "This has extra spaces and a line break in the source."
```

The one common exception: the `<pre>` element preserves whitespace and line breaks exactly as written — used for displaying code samples or any text where exact formatting genuinely matters.

Consistent Indentation — A Convention, Not a Rule

Chapter 1 already introduced the idea that browsers don't require indentation at all, but consistent indentation makes nested structure genuinely readable to a human. Most projects pick either 2 or 4 spaces and apply it consistently — the specific number matters far less than actually being consistent throughout a project.

Why "It Works in My Browser" Isn't the Same as Valid

Browsers' forgiving error-recovery behaviour means genuinely broken HTML — an unclosed tag, a missing required attribute, improperly nested elements — often still renders something acceptable-looking, hiding the underlying problem until it causes a real issue (often in a *different* browser, or after a future edit makes the existing error finally matter).

Using the W3C Validator

The W3C Markup Validation Service checks a page (by URL or pasted code) against the official HTML specification and reports every actual error and warning, with line numbers.

EXAMPLE VALIDATOR OUTPUT

ERROR, LINE 12

Element not allowed as child of element <p> in this context

ERROR, LINE 27

Attribute "src" without an attribute value (img element missing required value)

WARNING, LINE 1

The character encoding was not declared, defaulting to UTF-8

Common Validation Mistakes

Unclosed tags

A <p> or never closed — browsers often guess where it should end, but not always where you actually intended.

Improper nesting

A block element like placed inside a <p> — technically invalid, since <p> is only meant to contain inline content.

Missing required attributes

An <img> with no alt (Chapter 5), a <label> with no for, an <a> with no href.

Duplicate IDs

The same id attribute used on more than one element — IDs must be genuinely unique within a page (Chapter 1).

RUNNING THE VALIDATOR OCCASIONALLY DURING DEVELOPMENT CATCHES PROBLEMS EARLY

Validating only at the very end of a project means discovering a long list of accumulated issues all at once. Running it periodically during development — especially after adding any genuinely new structure, not just text content — catches problems while the relevant code is still fresh and easy to fix.

CHAPTER 10 QUICK REFERENCE

- **<!-- comment -->** — never visible to visitors, but visible in page source; never put sensitive info here
- **Whitespace collapses** to a single space when rendered, except inside <pre>

- **Consistent indentation** is purely for human readability — pick 2 or 4 spaces, apply consistently
- **Browsers' error tolerance** hides real bugs — "renders fine" isn't the same as "valid"
- **W3C Markup Validation Service** — checks real specification compliance, with line numbers
- **Common errors:** unclosed tags, improper nesting, missing required attributes, duplicate IDs
- **Validate periodically during development**, not just once at the end
- **Next chapter:** special characters & entities

CHAPTER 11: SPECIAL CHARACTERS & ENTITIES

COURSE 1 · CH 11

Special Characters & Entities

Why < and & need special handling, and the entity syntax that lets you display them safely

A handful of characters have special meaning to HTML's parser itself — using them directly in regular text content causes the browser to misinterpret what's actually being said. Entities solve this: a special escape syntax that displays a character literally, without the browser trying to interpret it as markup.

Why This Is Necessary at All

```
// Trying to literally write "5 < 10" in a paragraph...
<p>5 < 10 is true</p>

// The browser sees "<" and assumes a NEW TAG is starting,
// since "<" is the literal character that opens every tag.
// Result: broken, unpredictable rendering.
```

`<` signals the start of a tag; `&` signals the start of an entity (covered in this chapter itself). Using either character literally in regular text content confuses the parser — entities exist specifically to write these characters safely.

The Essential Entities

ENTITY	DISPLAYS AS	WHY IT'S NEEDED
`<`	<	Opens every HTML tag — must be escaped to display literally
`>`	>	Closes every HTML tag — escaped for consistency, though less strictly required than <

ENTITY	DISPLAYS AS	WHY IT'S NEEDED
`&`	&	Opens every entity, including itself — must be escaped to display a literal ampersand
`&quot;`	"	Needed inside an attribute value itself wrapped in double quotes
`&apos;`	'	Needed inside an attribute value wrapped in single quotes
` `	(non-breaking space)	A space that prevents a line break at that exact point — covered further below

```
// Written correctly, with entities
<p>5 &lt; 10 is true</p>
// Renders as: 5 < 10 is true

<p>Terms &amp; Conditions</p>
// Renders as: Terms & Conditions
```

Named, Decimal, and Hexadecimal Entities

Every entity has a named form (easier for a human to read in source) and equivalent numeric forms — useful for characters that don't have a commonly memorised name.

```
// Three equivalent ways to display a copyright symbol
&copy;       // named
&#169;       // decimal numeric
&#xA9;       // hexadecimal numeric
// All three render as: ©
```

Common Practical Entities

```
&copy;   // © copyright symbol
&trade;  // ™ trademark symbol
&reg;    // ® registered trademark
&mdash;  // — em dash, for a strong break in a sentence
&ndash;  // – en dash, for ranges like "pages 10&ndash;20"
```

```
&hellip; // … ellipsis
&pound;  // £ pound sterling symbol
&euro;   // € euro symbol
```

— The Non-Breaking Space

A regular space allows the browser to break a line at that point if needed for wrapping. ` ` behaves like a space visually but specifically prevents a line break there — keeping two words glued together on the same line.

```
// Without nbsp, "10" and "MB" could end up on separate lines if the text wraps
<p>File size: 10 MB</p>
```

USE SPARINGLY, FOR GENUINELY MEANINGFUL PAIRS

Good uses: keeping a number and its unit together (10 MB), a title and initial (J. R. Tolkien). Overusing it to force specific visual spacing throughout a page is generally better handled with CSS — entities solve a character-encoding problem, not a general layout one.

What About Unicode Characters Like 日本語 or 水?

Modern HTML documents, declared with UTF-8 encoding (the now near-universal standard), can include almost any character — emoji, Japanese kanji, accented letters — directly in the source text, with no entity needed at all. Entities are specifically for the small set of characters that have special meaning to HTML's own parser (`<`, `>`, `&`, quote characters inside matching attribute quotes), not a general mechanism for "any character that isn't plain English."

NEVER SKIP ESCAPING & AND < WHEN DISPLAYING USER-SUBMITTED CONTENT

If a website ever displays text a user typed (a comment, a username, a search query) back on the page without escaping these characters first, a malicious user could submit text containing real HTML/script tags that the browser then executes — this is the basis of cross-site scripting (XSS) attacks, covered properly in the browser security course's CSP chapter. Always escape user-submitted content before inserting it into a page.

CHAPTER 11 QUICK REFERENCE

- **< / > / &** — escape these whenever the literal <, >, or & character is genuinely intended in text
- **" / '** — needed inside an attribute value matching the surrounding quote style
- **Named vs numeric (© / ©)** — equivalent forms; named is more readable in source
- ** ** — a space that prevents line-wrapping at that point; use sparingly, for meaningful pairs
- **UTF-8 encoded documents** can include non-English characters and emoji directly — entities are only for parser-significant characters
- **Always escape user-submitted content** before displaying it — the basis of XSS prevention
- **Next chapter (final, Course 1):** putting it all together — building a simple multi-page site structure

CHAPTER 12: PUTTING IT ALL TOGETHER

COURSE 1 · CH 12 · FINAL CHAPTER · PROJECT

Putting It All Together

Building a real, simple multi-page site structure using everything covered across this Fundamentals course

This final chapter doesn't introduce anything new — it's a worked, end-to-end example combining every element from Chapters 1–11 into a genuine, if small, multi-page website structure, exactly as a real project would actually be organised.

Planning the File Structure

```
my-site/
├── index.html ← homepage
├── about.html
├── contact.html
└── images/
    └── logo.png
```

`index.html` is the conventional name for a folder's default page — most web servers serve it automatically when a visitor requests just the folder itself (`/` or `/about/`), without needing the filename in the URL.

A Complete, Realistic Page

This single example deliberately uses something from nearly every chapter in this course:

```
<!DOCTYPE html> // Ch1
<html lang="en"> // Ch1
<head>
  <meta charset="UTF-8">
  <title>About &amp;amp; Contact — My Site</title> // Ch1, Ch11
</head>
```

```
<body>

  <header> // Ch8
    <h1>My Site</h1> // Ch2
    <nav> // Ch8
      <ul> // Ch3
        <li><a href="/">Home</a></li> // Ch4
        <li><a href="/about.html">About</a></li>
        <li><a href="/contact.html">Contact</a></li>
      </ul>
    </nav>
  </header>

  <main> // Ch8
    <article> // Ch8
      <h2>About Us</h2>
      <img src="images/logo.png" alt="Our company logo"> // Ch5
      <p>We are a <strong>small team</strong> building things.</p> // Ch2
    </article>
  </main>

  <footer> // Ch8
    <p>&copy; 2026 My Site</p> // Ch11
  </footer>

</body>
</html>
```

Sharing Structure Across Pages

A real multi-page site repeats the same header/nav/footer on every page — at this stage in pure HTML, that means copying the same markup into each file. The repetition this creates is real, and it's exactly the motivation for templating systems and components covered in the website rebuild courses (FastAPI+Jinja2, Next.js) — this Fundamentals course intentionally stops short of that, since the goal here is understanding the raw building blocks first.

> **KEEPING NAVIGATION LINKS CONSISTENT ACROSS COPIED FILES IS A GENUINE MAINTENANCE CHALLENGE**

With three pages, manually keeping the <nav> identical everywhere is manageable. With thirty pages, it becomes a real liability — exactly the problem templating and component systems exist to solve, beyond the scope of pure HTML alone.

Final Review Checklist

- [x] **Doctype, html lang, head, body all present**
  The complete skeleton from Chapter 1, on every page.
  Chapter 1
- [x] **Heading levels form a logical outline, never skipped for sizing**
  Chapter 2
- [x] **Internal links are relative, not hardcoded absolute URLs**
  Chapter 4
- [x] **Every image has meaningful alt text (or an explicit empty alt for decorative images)**
  Chapter 5
- [x] **Page structure uses semantic landmarks, not div soup**
  Chapters 8-9
- [x] **Special characters are properly escaped, especially anywhere user content might appear**
  Chapter 11
- [x] **Run through the W3C validator before considering it genuinely done**
  Chapter 10

CHAPTER 12 QUICK REFERENCE — AND COURSE WRAP-UP

- **index.html** — conventional default filename, served automatically for a bare folder URL
- **A real multi-page site repeats header/nav/footer across files** — manageable at small scale, a real problem at large scale

- **Templating/components** (covered in the website rebuild courses) exist specifically to solve that repetition — intentionally out of scope here
- **Final checklist:** doctype/skeleton, logical headings, relative internal links, meaningful alt text, semantic landmarks, escaped special characters, validated markup
- **Course 1 recap:** document structure (Ch1) → text (Ch2) → lists (Ch3) → links (Ch4) → media (Ch5) → tables (Ch6) → forms (Ch7) → semantic HTML (Ch8) → divs/spans (Ch9) → comments/validation (Ch10) → entities (Ch11) → real project (Ch12)
- **Next:** Course 2 — HTML Intermediate (forms in depth, accessibility, meta tags, web components intro, and more)