Modern web development with HTML & CSS

2026 Edition

grimy86

Contents

Modern web development with HTML & CSS

Introduction

Web languages

- HTML, **HyperText Markup Language**: defines a web page's structure and content.
- CSS, **Cascading Style Sheets**: defines a web page's layout and style.
- JavaScript: makes the web page interactable and dynamic.

HTTP protocol

The **HyperText Transfer Protocol (HTTP)**, defines how hosts will exchange web page information. HTTP is a request & response protocol between a client and a server. When you visit a website, your browser (the client) sends a request to a server. The server then replies with files like HTML, images, etc.

HTTP is fundamentally platform-independent, and host devices will be able to communicate over the web, regardless of operating system or browser.

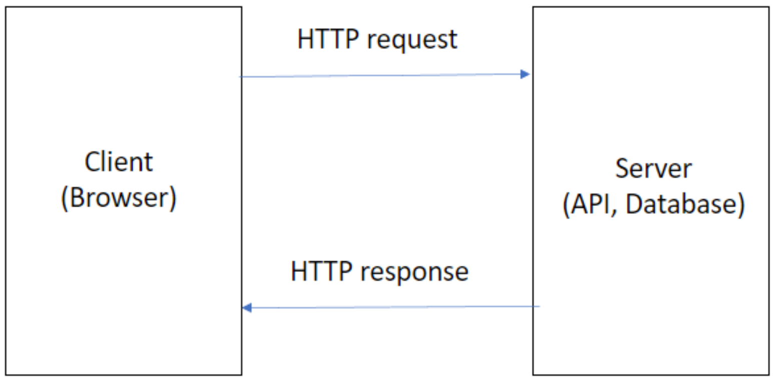

Figure 1: img.png

URL's

URL, **Uniform Resource Locator**: is a way to locate a resource like a webpage, image, file, etc. in the form of an address or path.

URL structure:

- Scheme / Protocol: HTTP, HTTPS, FTP, file, etc.
- Host: the domain name or the server's IP address
- TCP port: by default, this is 80 for HTTP and 443 for HTTPS
- Path: this could be any path the server has made public
- Query: this is non-hierarchical data, a question mark followed by a key-value-pair and divided by an ampersand
- Fragment-identifier: This is a subdocument or specific part of the document

Example: `https://img.shields.io:443/badge/C-00599C?logo=c&logoColor=fff&style=for-the-badge`

Dissected example:

- `https://`
- `img.shields.io`
- `:443`
- `/badge/C-00599C`
- `?logo=c&logoColor=fff&style=for-the-badge`
- no fragment-identifier

Integrated Development Environments (IDEs)

IDEs

An IDE is a program or tool a developer uses to develop other programs or tools, like web pages or applications.

Basic web development doesn't have complicated processes like compiling binaries. A glorified text-editor will suffice.

Most common IDEs for web development:

- Visual Studio Code (VS Code)
- JetBrains WebStorm

HTML structure

To start with HTML, define a file with the `.html` extension. By most developers, the main page is defined as `index.html`.

We need to define `<!doctype html>` to let the browser know that this file uses **HTML5**. Next, we define `<html lang="en">` to represent the **root** of our file.

With the `lang` attribute, we indicate the language the web page is written in. Next, we define the `<head>` and `<body>` of our page within the `</html>` closing tags. The `<head>` contains metadata and the `<body>` contains the page's content like text, images, videos, etc. Now all we're left to do is make sure to add all closing tags in the right order.

An empty web page:

```html
<!doctype html>
<html lang="en">
  <head>
  </head>
  <body></body>
</html>
```

Document Object Model (DOM)

When a browser loads an HTML file, it converts the HTML structure into a DOM tree. The DOM **represents the page as a hierarchy of nodes**, where each HTML element becomes an object the browser can work with.

This structure allows the browser to:

- Render the page visually
- Access and modify elements using JavaScript
- Apply styles using CSS

The `<html>` element is the root of the DOM, with `<head>` and `<body>` as its direct children. All elements inside the page are part of this tree structure, as shown in the image above.

By working with the DOM, developers can dynamically change content, structure, and behavior of a web page without reloading it.

Metadata & SEO

Like we've mentioned before the `lang` attribute in the opening `<html lang="en">` tags provides metadata. The entire `<head>` of the file is also meant to provide metadata.

Metadata is data that can be used or processed by browsers, search engines, accesability tools, etc.

Examples for browsers:

- `<meta charset="utf-8" />`

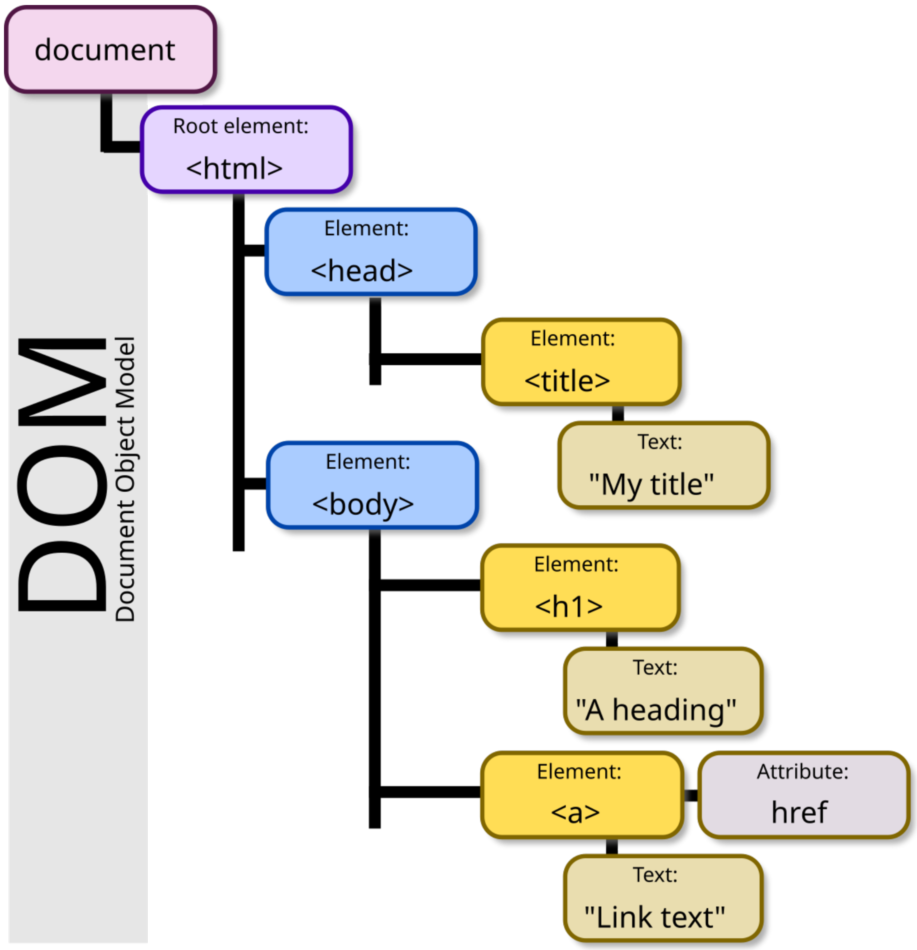

Figure 2: DOM.png

- `<meta name="viewport">`

Examples for **Search Engine Optimization (SEO):**

- `<meta name="description">`
- `<meta name="author">`
- `<link>` to a `.css` file
- `<title>` of the browser tab

Elements, tags and nesting

An element is essentially part of a web page. They're usually opened and closed but some elements are **self-closing**.

Some elements can also contain other elements, this is referred to as **nesting**. Nesting is where indentation and IDE tools like formatters become important.

Examples:

- `<p>Some text</p>`
- `<img src="img/SomeImage.jpg"/>`
- `<br>`
- `<head><meta charset="utf-8"/><title>SomeTitle</title></head>`

Attributes

Each element can have attributes within its **opening tags**. Some attributes are global and some are specific to the element. Attributes again provide more data for the element, like the **source** of an image or the **hypertext reference** (link) to a resource.

Make sure when using attributes that you use regular `""` quotes and not the typographic `""` quotes, as those are processed by browsers in a different way. Their usage may lead to errors, usually they're copied from websites or Word-documents.

Examples:

- `src="text.gif"`
- `href=index.html`
- `class="container"`

CSS attributes & inline CSS

There's also an amount of CSS attributes that an element can use. These usually look like `style="border:1px solid red;"`.

Using the style attribute within element opening tags like this is known as **inline CSS**. It is highly discouraged and considered malpractice.

CSS styles are supposed to be used in seperate files and referenced to through the `<head>` metadata.

Comments

Comments are not processed by the browser and give more information about your source code.

Example: `<!-- This is a comment -->`

Special characters & entity names or numbers

Because the browser assigns special meaning to certain characters, we need an alternative way to type them when we want to display them on a web page.

For example, `<` and `>` are used to mark the beginning and end of an HTML element. To display these characters literally, we use **entity names or numbers**.

There's entity names or numbers for every single character in the ASCII table.

See examples

Conventions & code validation

Sometimes your HTML source code might seem like it follows all the rules when it doesn't.

Some HTML5 requirements:

- Including `<!DOCTYPE html>`
- Including `<meta charset="utf-8">`
- Correctly ordering nested elements
- All HTML elements and attributes must be defined in lowercase
- Always but values between double quotation marks
- Elements with a closing tag must be closed properly
- Standard text should always be placed in a paragraph element

A good way to find out if your source code runs properly across multiple browsers is to **validate** it. Through the process of validation, diagnosing and editing your source code you achieve a **standardised and browser-independent** source.

Element layout

Block, inline & inline-block elements

`Block`: elements start on a new line and take up the full available width by default:

- `<p>`
- Headers like: `<h1>`, `<h2>`, `<h3>`, etc.
- `<div>`
- `<pre>`
- `<blockquote>`
- `<ul>`
- `<ol>`

`Inline`: elements do not start on a new line and only take up as much width as their content. You cannot set width or height:

- `<code>`
- `<q>`
- `<a>`
- `<span>`
- `<img/>`
- `<input/>`
- `<br/>`
- `<wbr/>`
- Text formatters like: `<strong>`, `<sup>`, `<sub>`, `<b>`, `<i>`

`Inline-block`: elements do not start on a new line, but you can set width and height like a block element.

HTML semantic structural elements

Historically `<div>` elements were used to make up the structure of a page. As of today we have **semantic** structural elements that are **more meaningful** as to their content and structure.

Semantic structural elements:

- Headings: `<header>`
- Navigation blocks: `<nav>`
- Main content: `<main>`
- Sections: `<section>`
- Articles: `<article>`

- Footers: `<footer>`
- Aside information: `<aside>`

Practical example of a page:

```html
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <header>
      <h1>Page title</h1>
      <p>Page description</p>
    </header>
    <nav>
      <ul>
        <li><a href="link1.html">Link 1</a></li>
        <li><a href="link2.html">Link 2</a></li>
        <li><a href="link3.html">Link 3</a></li>
      </ul>
    </nav>
    <section>
      <header>
        <h2>Section title</h2>
      </header>
      <p>Section contents</p>
      <footer>
        <p>Section footer</p>
      </footer>
    </section>
    <aside>
      <p>Aside information</p>
    </aside>
    <footer>
      <p>Page Footer</p>
    </footer>
  </body>
</html>
```

Hyperlinks with the anchor element

We can user hyperlinks with the `<a href="...">string</a>` element. The `a` here stands for **anchor** while the `href` attribute means **hypertext reference**.

There's another commonly used attributed: `target="..."` with two common arguments:

- `_self`, open the reference in the current page (default setting)
- `_blank`, open the reference in a new blank page

Absolute, relative references & internal references

An **absolute** link points to a specific document on a web server using a full URL.

Example: `<a href="http://www.example.com/page2.html">Page 2</a>`

A **relative** link points to another file relative to the location of the current file.

Examples:

- Same working directory: `<a href=".page1.html">Page 2</a>`
- One directory down: `<a href="./pages/page2.html">Page 2</a>`
- One directory up: `<a href="../index.html">...</a>`

You'll notice a trend in the examples above where `./` is used explicitly. The browser assumes the current folder by default, so `./` is optional.

An internal link is used to navigate to a **specific position within the same document** or another document.

An internal link always consists of two **required** parts:

- An anchor, which identifies an element by adding an `id` attribute (e.g. id="someId").
- A hyperlink that uses a hashtag, `#` to point to that anchor (e.g. href="#someId").

Example:

```html
<h2>Index</h2>
<ul>
  <li><a href="#part1">part 1</a></li>
  <li><a href="#part2">part 2</a></li>
</ul>
```

```html
<article id="part1">
  <h3>part 1: Intro</h3>
</article>
<article id="part2">
  <h3>part 2: Structure</h3>
</article>
```

Lists, list items & nested lists

Ordered lists

An ordered list gives a numerical order to its list items.

Example:

```html
<ol>
  <li>item A</li>
  <li>item B</li>
  <li>item C</li>
  <li>item D</li>
  <li>item D</li>
</ol>
```

Unordered lists

An ordered list gives no order to its list items and uses symbols instead.

Example:

```html
<ul>
  <li>item A</li>
  <li>item B</li>
  <li>item C</li>
  <li>item D</li>
</ul>
```

Nested lists

Nested lists are lists that contain other lists within themselves. A `<ul>` or `<ol>` can only list items, `<li>`. To create a nested list you have to create a new `<ul>` or `<ol>` within an existing `<li>`.

Example:

```
<ol>
  <li>item A
    <ul>
      <li>item B1
        <ul>
          <li>item B2.1</li>
          <li>item B2.2</li>
        </ul>
      </li>
      <li>item B2</li>
    </ul>
  </li>
  <li>item B</li>
  <li>item C</li>
</ol>
```

Media

Images

To add an image to a web page, you use the `<img>` element. This element has a `src="..."` attribute that specifies the image URL and an `alt="..."` attribute that provides alternative text.

The alt text serves the following purposes:

- **Fallback**, the alt text will be desplayed in case the image is missing, broken, etc.
- **Accessiblity**, screen readers use it so visually impaired or blind users understand what the image represents.
- **SEO**, search engines use the alt text to better understand what the image is about, which can help the image and/or page rank higher in search results.

Example: `<img src="https://picsum.photos/id/237/200/300" alt="cute dog image">`

Image formats

When working with images on the web, it is important to choose the right file format. Each format has advantages and disadvantages depending on the type of image and its intended use.

The three most commonly used formats are GIF, JPEG, and PNG.

- **GIF**, supports a limited number of colors but allows animation and simple transparency.
- **JPEG**, uses lossy compression, resulting in smaller file sizes and is best suited for photos.
- **PNG**, uses lossless compression, supports transparency, and is ideal for images with many colors or sharp edges.

Feature	GIF	JPEG	PNG
Number of colors	256	16 million	16 million
Compression	Lossless	Lossy	Lossless
File size	Larger	Smaller	Smaller
Download time	Longer	Shorter	Shorter
Decompression time	Shorter	Longer	Longer
Animation	Recommended	Not recommended	Not recommended
Transparency	Supported (GIF89a)	Not supported	Supported

Recommendations:

- For logos, use GIF
- For computer drawings, use GIF
- For photos, use JPEG or PNG

Grouping images

With the `<figure>` and `<figcaption>` element you can group an image with a caption.

```
<figure>
  <img src="https://picsum.photos/200/100" alt="Some image">
  <figcaption>This is some image.</figcaption>
</figure>
```

Images as a reference

You can also make an image have a reference.

Example

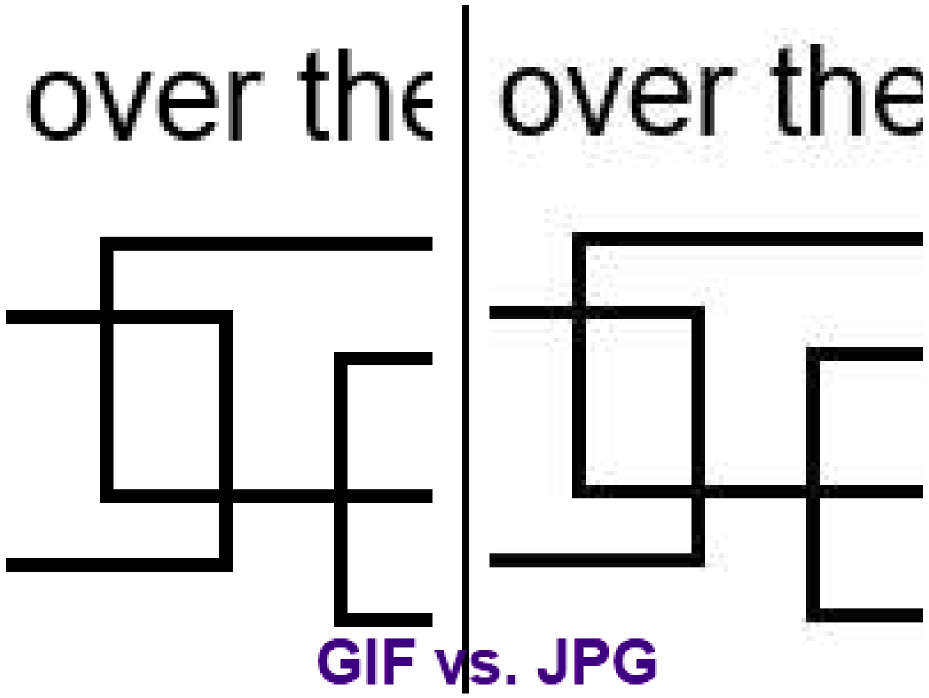

Figure 3: GIF.png

```html
<a href="https://picsum.photos/200/100">
  <img src="https://picsum.photos/200/100" alt="Some image">
</a>
```

Iframes

With the `<iframe>` element you can render content from other webpage's within your own web page.

Example: `<iframe width="560" height="315" src="https://www.youtube-nocookie.com/embed/dQw4w9WgXcQ?si=LfNzY879bUEiJKzg" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture; web-share" referrerpolicy="strict-origin-when-cross-origin" allowfullscreen></iframe>`

CSS

CSS stands for Cascading Style Sheets. A cascade is what a waterfall does, it **flows from top to bottom**. This means **CSS styles are applied from top to bottom**, where the last applied rule determines the final appearance.

Syntax: selectors, properties & keys

A line of CSS code consists of two parts:

- **Selector:** specifies the element or elements to which the style is applied.
- **Declaration block:** a block surrounded by curly braces `{}` that contains one or more rules.

Each rule inside the declaration block consists of a **property (key)** and a **value**, separated by a colon `:` and ending with a semicolon `;`.

Below we see that the p **selector** styles all `<p>` elements using two rules:

- `color: red;`, makes the text color red.
- `font-weight: bold;`, makes the text bold.

Applying the same style to multiple elements is also possible. It's done by selecting them, e.g: `h1, h2`. This means that all `<h1>` elements and all `<h2>` elements will receive the same style.

Examples:

```
/*
    This is a multi-line comment in CSS.
*/

p {
  color: red;
  font-weight: bold;
}

h1,
h2 {
  color: blue;
  font-family: Arial, sans-serif;
}
```

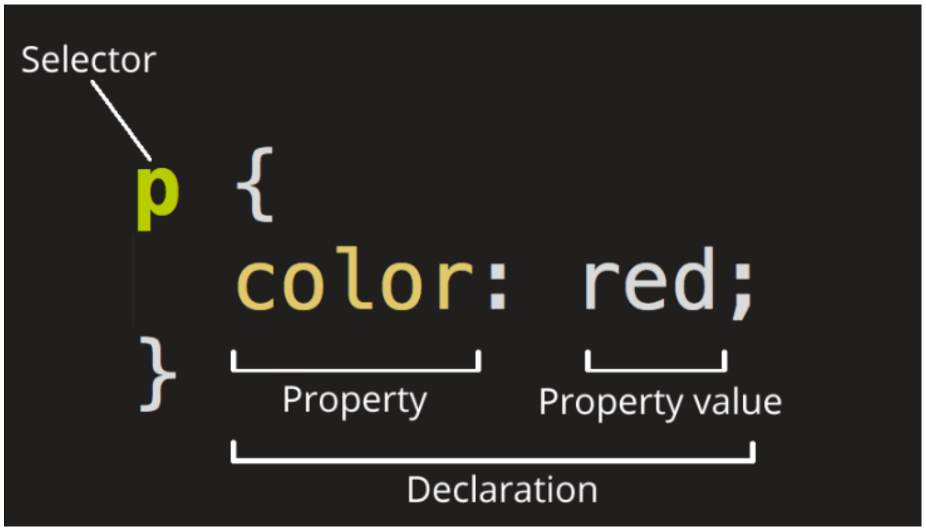

Figure 4: css_syntax.png

Inline, internal & external CSS

We've already discussed inline CSS and came to the conclusion that it's malpractice. There's another form of malpractice named **internal CSS styling**. With this method we place a `<style>` element within the document `<head>`.

The best practice is to create a seperate styles file like `style.css`, then link to it within the document `<head>`.

Example internal CSS styling:

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

    <!-- Internal CSS -->
    <style>
      p {
        color: red;
        font-size: 12pt;
      }
    </style>
  </head>
  <body>
    <p>Some text</p>
  </body>
</html>
```

Example external CSS styling:

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

    <!-- External CSS reference -->
    <link rel="stylesheet" href="./style.css" />
  </head>
  <body>
    <p>Some text</p>
  </body>
</html>
```

The `rel` attribute

The `rel` attribute in the `<link>` tag specifies the **relationship** between the current document and the linked file. In the example above, `stylesheet` means that the file is used to define the styling of the HTML page.

CSS selectors

Tag selector

A `tag` selector selects all elements of a specific type of element.

The `p` selector selects all `<p>` elements:

```css
p {
  color: red;
}
```

Class selector

A `class` selector selects all HTML elements that have a specific value in the class attribute. In CSS, classes are defined using a dot, `.` before the class name. Best practice is to always **use very descriptive crystal-clear class names**.

For example, the selector `.important` selects all HTML elements with the attribute `class` important.

HTML class definition:

```html
<div class="important">
    This is an important element.
</div>
```

CSS class selector:

```css
.important {
  font-weight: bold;
  color: red;
}
```

Combination of multiple classes is also possible in HTML, this makes it possible to have an element inherit multiple styles:

```html
<p class="important highlighted">
  This is an important highlighted element.
</p>
```

CSS class selector:

```css
.important {
  font-weight: bold;
  color: red;
}

.highlighted {
  background-color: yellow;
}
```

Tag with class selector

Suppose you want to style only the `<p>` elements that have the class important, but not a `<div>` element with the same class.

You can define a more specific selector by combining a tag selector with a class selector. Only elements that match both the tag and the class attribute will be selected. In the example below only the `<p>` tag of this specific class will be targeted.

HTML:

```html
<p class="important">This is an important paragraph.</p>
<div class="important">This is an important div element.</div>
```

CSS:

```css
p.important {
  font-weight: bold;
  color: red;
}
```

ID selector

An `ID` selector selects a specific element with a unique `id`. References to IDs are defined using a hash `#` before the ID name. For example, the selector `#special` selects the element with the ID special.

HTML:

```html
<p id="special">This is a special element.</p>
```

CSS:

```
#special {
  font-weight: bold;
  color: green;
}
```

You will not get an error if multiple elements use the same ID. It is your **responsibility** as a developer to ensure that each ID appears only once.

This rule only applies per page. For example, **both index.html and about-us.html can contain an element with the ID special**. This is not a problem, because IDs must only be unique within a single page.

In some companies, the use of ID selectors is discouraged because they are not reusable. In those cases, **IDs are used only as anchors for in-page navigation**, and all styling is done using class selectors. Best practice is to avoid styling using ID selectors.

Wildcard selector

A `wildcard` selector selects **all elements** on an HTML page. This is done using an asterisk `*` as the selector.

Use this selector with **caution** as outside of general page styling, the use of the wildcard selector is **discouraged**. It can lead to **unpredictable results and conflicts** with other styles.

CSS:

```
* {
  color: blue;
}
```

Contextual selector

`Contextual` selectors select elements based on their **relationship** to other elements. This is useful when you want to style elements that appear within a specific context.

For example, to select only `<p>` elements that are inside a `<div>` element, you use the following selector:

```
<p>This is not selected</p>
<div> This is a div.
  <p>This is selected.</p>
</div>
```

```css
div p {
  color: green;
}
```

Example, select all `<li>` inside a `<ul>` (and not the `<li>` elements inside an `<ol>`):

```css
ul li {
  color: orange;
}
```

Example, select all `<li>` elements that are inside a `<ul>` element with the class `menu`:

```css
ul.menu li {
  color: purple;
}
```

Attribute selector

An `attribute` selector selects elements **based on their attributes**. This is useful for styling elements that have specific attributes or specific attribute values. In CSS, attributes are selected using square brackets `[]`.

For example, you can add a green border to all images that have an alt attribute:

```css
img[alt] {
  border: 2px solid green;
}
```

The opposite is also possible:

```css
img:not[alt] {
  border: 2px solid red;
}
```

You can add a green border to all links whose href attribute **starts with** `https://`. This allows you to verify that all links point to a secure website (no `http://` links):

```css
a[href^="https://"] {
  border: 2px solid green;
}
```

If you only want to style links whose href attribute **ends with** `example.com`, you can use this selector:

```
a[href$="example.com"] {
  border: 2px solid blue;
}
```

Pseudo-class selectors

A `pseudo-class` is a special selector that selects an element **based on a state or condition**. This is useful for applying styles in specific situations, such as when a user hovers over an element or when a form element is in a certain state.

HTML:

```
<a href="https://www.github.com" target="_blank">GitHub</a>

<ul>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
  <li>Item 4</li>
  <li>Item 5</li>
</ul>
```

CSS:

```
/* Style links when the mouse hovers over them
 *
 * a:hover does not apply to already visited links by default.
 * To ensure the hover style always works, we target both
 * a:link:hover and a:visited:hover.
 */
a:link:hover,
a:visited:hover {
  color: darkorange;
}

/* Style visited links */
a:visited {
  color: deeppink;
}

/* Style unvisited links */
a:link {
  color: darkgreen;
}

/* Style the first child of an element */
ul li:first-child {
  font-weight: bold;
}

/* Style the last child of an element */
ul li:last-child {
  font-style: italic;
}

/* Style the second child of an element */
ul li:nth-child(2) {
  color: green;
}

/* Style all odd children */
ul li:nth-child(odd) {
  background-color: lightgray;
}

/* Style all even children */
ul li:nth-child(even) {
  background-color: lightblue;
}
```

Recap with practical examples

1. `ul.menu`, styles the
   itself:

```
<ul class="menu">...</ul>     selected
<ol class="menu">...</ol>     not selected
```

2. `.menu li`, styles `<li>` elements inside **ANY** `.menu` element:

```
<div class="menu">
  <li>Item</li>     selected
</div>

<ul class="menu">
```

```
  <li>Item</li>     selected
</ul>
```

3. `ul.menu li`, styles `<li>` elements **inside** `<ul class="menu">`:

```
<ul class="menu">
  <li>Item</li>     selected
</ul>

<div class="menu">
  <li>Item</li>     not selected
</div>
```

Selector summary

Selector summary

Selector type	Selector	What it selects
Tag selector	`li`	All `<li>` elements
Class selector	`.menu`	All elements with class `menu`
ID selector	`#special`	The element with id `special`
Tag + class selector	`ul.menu`	`<ul>` elements with class `menu`
Descendant selector	`.menu li`	`<li>` elements inside any element with class `menu`
Contextual selector	`ul.menu li`	`<li>` elements inside `<ul class="menu">`
Multiple class selector	`.menu.li`	Elements that have both classes `menu` and `li`
Wildcard selector	`*`	All elements on the page
Attribute selector	`img[alt]`	`<img>` elements that have an `alt` attribute

Selector type	Selector	What it selects
Negated attribute selector	`img:not([alt])`	`<img>` elements without an `alt` attribute
Attribute starts-with	`a[href^="https://"]`	`<a>` elements whose `href` starts with `https://`
Attribute ends-with	`a[href$="example.com"]`	`<a>` elements whose `href` ends with `example.com`
Pseudo-class (hover)	`a:hover`	`<a>` elements when hovered
Pseudo-class (visited)	`a:visited`	`<a>` elements that have been visited
Pseudo-class (link)	`a:link`	`<a>` elements that have not been visited
Pseudo-class (first child)	`li:first-child`	The first `<li>` inside its parent
Pseudo-class (last child)	`li:last-child`	The last `<li>` inside its parent
Pseudo-class (nth child)	`li:nth-child(2)`	The second `<li>` inside its parent
Pseudo-class (odd)	`li:nth-child(odd)`	All odd `<li>` elements inside their parent
Pseudo-class (even)	`li:nth-child(even)`	All even `<li>` elements inside their parent

Units

In CSS, you can use multiple units (for example, to define the width of an element). Below are some of the most commonly used units.

Unit	Description
`px`	Pixels, an absolute unit based on screen resolution. `1px` corresponds to one screen pixel.
`pt`	Points, mainly used for print. `1pt = 1/72 inch`. Commonly used in print media such as PDFs.

Unit	Description
`%`	Percentage relative to the parent (containing) element. Often used for layouts and scalable elements.
`em`	Relative unit based on the font-size of the current element. For example, `2em` means twice the current font size.
`rem`	Root em. Relative to the font-size of the root element (usually `<html>`, commonly `16px`). `2rem = 2 × root font-size`.

Colors

Original CSS colors

Originally, CSS1 defined **16 color names**. In CSS2, the **color orange was added**. These color names are still valid in CSS3.

Color	Hex code
black	#000000
silver	#c0c0c0
gray	#808080
white	#ffffff
maroon	#800000
red	#ff0000
purple	#800080
fuchsia	#ff00ff
green	#008000
lime	#00ff00
olive	#808000
yellow	#ffff00
navy	#000080
blue	#0000ff
teal	#008080
aqua	#00ffff
orange (CSS2)	#ffa500

rebeccapurple

Later versions of CSS introduced **around 150 additional** color names, such as deeppink, lightblue, midnightblue, etc. There is also a special color name called `rebeccapurple (#663399)`. This color was added in memory of Rebecca Meyer, the daughter of web developer Eric Meyer, who passed away in 2014.

Hexadecimal colors

Besides color names, you can also define colors using **hexadecimal color codes**. These colors start with a hash `#` and are followed by six hexadecimal digits, for example `#993366`.

Hexadecimal digits include the numbers **0 to 9 and the letters A to F**, where:

Hex digit	Decimal value
0–9	0–9
A	10
B	11
C	12
D	13
E	14
F	15

A hexadecimal color is split into three parts: **red (R)**, **green (G)**, and **blue (B)**. Each color component is represented by **two hexadecimal digits**, meaning each component can have a value from **00 to FF** (**0 to 255** in decimal notation).

Examples

- `#FF0000` → 255 red, 0 green, 0 blue → pure red
- `#00FF00` → 0 red, 255 green, 0 blue → pure green
- `#0000FF` → 0 red, 0 green, 255 blue → pure blue
- `#FFFFFF` → 255 red, 255 green, 255 blue → white
- `#000000` → 0 red, 0 green, 0 blue → black
- `#663399` → 102 red, 51 green, 153 blue → rebeccapurple

HSL colors

HSL stands for **Hue (H)**, **Saturation (S)**, and **Lightness (L)**. It is an alternative way to define colors that is often more intuitive than RGB.

1. Hue ranges from 0–360 degrees:

- 0 = red
- 120 = green
- 240 = blue
- 360 = same as 0 (red)

2. Saturation ranges from `0%` (gray) to `100%` (full color)

3. Lightness ranges from `0%` (black) to `100%` (white)

Examples:

```
/* red */
color: hsl(0, 100%, 50%);

/* green */
color: hsl(120, 100%, 50%);

/* blue */
hsl(240, 100%, 50%)

/* white */
hsl(0, 0%, 100%)

/* black */
hsl(0, 0%, 0%)

/* rebeccapurple */
hsl(270, 50%, 40%)
```

Transparency (Alpha)

All color formats can include transparency, also called **alpha**.

Hexadecimal transparency:

- Add two extra hex digits at the end
- `00` → fully transparent
- `FF` → fully opaque

RGB transparency (RGBA):

- RGBA extends RGB with an alpha value from `0` to `1`.
- `rgba(0, 0, 0, 1)` → opaque
- `rgba(0, 0, 0, 0.5)` → 50% transparent
- `rgba(0, 0, 0, 0)` → fully transparent

HSL transparency (HSLA):

- HSLA extends HSL with an alpha value from `0` to `1`.
- `hsla(0, 0%, 0%, 1)` → opaque
- `hsla(0, 0%, 0%, 0.5)` → 50% transparent
- `hsla(0, 0%, 0%, 0)` → fully transparent

Text

Styling

CSS provides several properties to control the appearance of text, such as `font`, `size`, `weight`, `alignment`, and `spacing`.

CSS property	Description / Values
`font-family`	Name of the font
`font-style`	`normal` | `italic` | `oblique`
`font-variant`	`normal` | `small-caps`
`font-weight`	`normal` | `bold` | `bolder` | `lighter` | `100-700`
`font-size`	Keywords, length units (`px`, `em`, `%`, etc.)
`font`	Shorthand for multiple font properties

Example:

```css
p {
  font-family: Arial;
  font-style: italic;
  font-variant: small-caps;
  font-weight: bold;
  font-size: 16px;
  line-height: 1.5;
}
```

Shorthand `font` property example:

```css
p {
  font: italic small-caps bold 16px/1.5 Arial;
}
```

Generic font families & fallbacks

Generic font families can be used as a fallback when a specific font is not available on the user's system. It is the **developer's responsibility** to specify fallback fonts.

Note that fonts with multiple words containing spaces **must be wrapped in quotes**.

Generic family
`serif`
`sans-serif`
`monospace`
`cursive`
`fantasy`

Example:

```css
/* No quotes */
p {
  font-family: Arial, sans-serif;
}

/* Wrapped in quotes */
.important p {
  font-family: "Open Sans", Arial, sans-serif;
}
```

Text styling properties

CSS property	Description / Values
`direction`	Text direction: `ltr` | `rtl`
`line-height`	Line height: number with unit
`text-align`	`left` | `right` | `center` | `justify`
`text-decoration`	`none` | `underline` | `overline` | `line-through` | `blink`
`text-shadow`	`h-shadow v-shadow blur color`
`vertical-align`	`baseline` | `sub` | `super` | `top` | `middle` | `bottom` | `%`

Boxmodel

Every HTML element is rendered as a rectangular box made of four layers moving from inside to outside:

1. Content: the actual content (text, image, etc.)
2. Padding: transparent space inside the border and around the content
3. Border: the edge around the element
4. Marding: transparent space outside the element (spacing between elements)

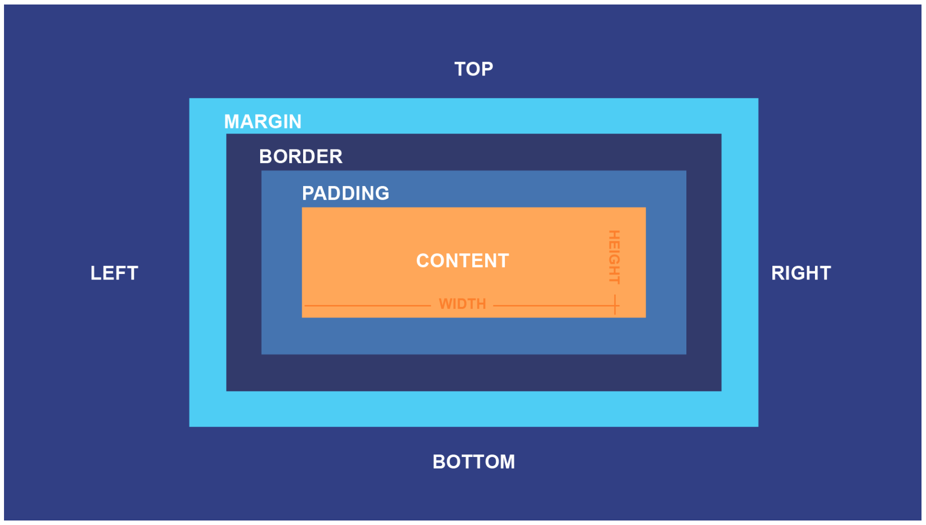

Figure 5: box_model.png

Padding, margin & the clockwise rotation

Padding is the space between the content and the border. **Margin** is the space outside the border.

Think of the padding property as a **clockwise** padding:

```
/* all sides */
p {
    padding: 10px;
    margin: 10px;
}

/* top/bottom - right/left */
p {
    padding: 10px 10px;
    margin: 10px 10px;
}

/* top - right/left - bottom */
p {
    padding: 10px 10px 10px;
    margin: 10px 10px 10px;
}

/* top - right - bottom - left */
```

```
p {
    padding: 10px 10px 10px 10px;
    margin: 10px 10px 10px 10px;
}
```

There's also seperate properties:

- `padding-top, padding-right, padding-bottom, padding-left`
- `margin-top`, `margin-right`, `margin-bottom`, `margin-left`

Border

The edge around an element.

Most common shorthand:

```
p {
  border: 10px dashed red;
}
```

There's also seperate properties:

- `border-width` → border size
- `border-style` → style of the border line
- `border-color` → color of the border line
- `border-radius` → rounded corners
- `border-image` → image border
- `box-shadow` → shadow around element

Style properties for block-level elements

CSS property	Value	Description
`float`	`left`	`right`
`clear`	`left`	`right`
`width`	length unit	Sets the width of the element
`height`	length unit	Sets the height of the element
`margin`	length values	Space **outside** the element
`padding`	length values	Space **inside** the element
`border`	width style color	Border around the element

Visual & border-related properties

CSS property	Description
`box-shadow`	Adds a shadow: color, x-offset, y-offset, blur, spread
`border-radius`	Rounds the corners of the element
`border-image`	Uses an image as a border (`url()`)

Display property

The display property controls how an element behaves and how it is laid out.

Value	Behavior
`block`	Takes full width, starts on a new line
`inline`	Takes only necessary width, stays inline
`inline-block`	Inline element with width and height support
`flex`	Enables flexible, responsive layouts
`grid`	Enables rows and columns
`none`	Element is hidden and takes no space

Style properties for list elements

CSS property	Value	Description
`display`	`block`	`inline`
`list-style-type`	`disc`, `circle`, `square`, `none`	Bullet styles
	`decimal`, `lower-roman`, `upper-roman`, `lower-alpha`, `upper-alpha`	Numbering styles
`list-style-image`	`url()`	Image used as bullet
`list-style-position`	`inside`	`outside`
`list-style`	shorthand	Combines type, image, and position

Practical tips

- Use `display: none` to remove elements entirely

- Prefer `inline-block` when you need inline layout with size control
- **Avoid heavy use of `float`** in modern layouts (Flexbox replaces it)

Origin and specifity

Origins & priority

The origin of a CSS rule **refers to where the style rule is defined**. CSS distinguishes between different origins, and these origins affect which styles are ultimately applied.

Priority	Origin	Description
1	**User agent (browser)**	Default styles applied by the browser (e.g. default margins on `<body>`, font sizes for headings).
2	**Author (Inline, Internal & External CSS)**	Styles written by the website developer in CSS files or `<style>` blocks.
3	**User (accessibility settings)**	Styles defined by the user, such as browser accessibility settings or user style sheets.
4	**User-defined (extensions)**	Styles added by users via browser extensions or custom overrides.

Specifity

Specificity determines which CSS rule is applied **when multiple rules target the same element**.

Specificity order (ranked highest to lowest):

1. Inline styles (style="…")
2. ID selectors
3. Class selectors
4. Tag selectors
5. Wildcard selector (*)

Higher specificity always beats lower specificity, regardless of order in the CSS file. Even though `.text` appears later, the **ID selector wins because it has higher specificity**:

```css
#unique-text {
  color: red;
}

.text {
  color: blue;
}
```

Specifity calculation & practical example

Specificity is written as **(a-b-c)**:

- a = number of ID selectors
- b = number of class selectors
- c = number of tag selectors

HTML:

```html
<p>Normal p-tag</p>
<p id="special">Special</p>
<p class="highlight">Highlighted</p>
<p class="highlight" id="special">Special and highlighted</p>
<p class="cool highlight">Cool and highlighted p-tag</p>
<span class="cool highlight">Cool and highlighted span-tag</span>
```

```css
p {
  color: red; /* Tag selector: 0-0-1 */
}

.highlight {
  color: blue; /* Class selector: 0-1-0 */
}

#special {
  color: green; /* ID selector: 1-0-0 */
}

p.highlight {
  color: purple; /* Tag + Class selector: 0-1-1 */
}
```

```css
#special.highlight {
  color: orange; /* ID + Class selector: 1-1-0 */
}

.cool .highlight p {
  color: pink; /* Class + Tag selector: 0-2-1 */
}
```

HTML Tables

Introduction

A table consists of several elements that together define its structure:

- `<table>`: defines the table
- `<tr>`, Table Row: a **row** in the table
- `<td>`, Table Data: a cell containing **data**

A `<td>` must always be inside a `<tr>`, and a `<tr>` must always be inside a `<table>`. For semantic clarity, we usually also add a `<tbody>` element. This is not required, but it makes the table structure clearer:

```html
<table>
  <tbody>
    <tr>
      <td>cell 1</td>
      <td>cell 2</td>
    </tr>
  </tbody>
</table>
```

Styling

Default

By default no table layout is visible, only the data. It is our responsability to add styling ourselves using CSS.

Borders

Border-collapse:

- `border-collapse: separate`: default, keeps borders separate
- `border-collapse: collapse`: merges adjacent borders into a single border, resulting in a cleaner and tighter table layout.

Border-spacing: controls the space between cells.

CSS:

```css
/* adds borders around each cell */
td {
  border: 2px solid black;
}

table {
  border: 4px solid deeppink;
  border-collapse: collapse;
  border-spacing: 10px;
}
```

Table headers using `<thead>` semantics

With a table head, we can add a header row to a table. This makes the data easier to understand. The table head is placed inside the semantic `<thead>` element.

Using `<thead>` is not required, but it improves the structure and readability of the table.

Using `scope` for accessibility

The `scope="..."` attribute has no visual effect on a table by itself. Its purpose is to help screen readers understand which data a `<th>` header applies to.

It tells assistive technologies whether a header describes:

- `scope="col"`: a column
- `scope="row"`:a row

Table titles with `caption`

You can add a title to a table using the `<caption>` element. You can change the position of the caption using the `caption-side` property.

Table structure

So far, we have used `<thead>` and `<tbody>` to define the structure of a table. There is one more semantic structural element we can use: `<tfoot>`.

The `<tfoot>` element is used to define the **footer** of a table, for example to display:

- totals
- summaries
- calculated values

Column styling using `colgroup` & `col`

With the `<colgroup>` and `<col>` elements, you can define styles for entire columns in a table. This is useful when you want to:

- assign a fixed width to a column
- apply a background color or other styles to a whole column at once

Unlike styling `<td>` or `<th>`, column styling applies before cells are rendered, making it ideal for **layout-related rules**.

Cell spanning using `colspan` & `rowspan`

The `colspan` and `rowspan` attributes **do not style or group** rows or columns. These attributes affect **table layout**, not styling or semantics.

They only control how table cells **span across** rows or columns:

- `colspan`: makes a `<td>` or `<th>` span across an amount of columns.
- `rowspan`: makes a `<td>` or `<th>` span across an amount of rows.

Table layout

The `table-layout` property controls **how the browser calculates column widths** in a table. It is especially useful when a table has a fixed width.

`table-layout: auto:`

- This is the **default** behavior
- The browser inspects all cell content
- Columns become wider if they contain wider content

- Layout calculation is slower but more flexible

`table-layout: fixed:`

- Column widths are calculated only from the table width
- Content does not affect column width
- Columns are distributed evenly (unless widths are explicitly set)
- Layout calculation is faster
- Overflowing content may wrap or overflow

Overflowing text in table cells

Sometimes a table cell does not have enough space to display its content. When this happens, text can **overflow** and **break the layout**.

CSS provides a couple of ways to handle this:

- **Breaking long words** using `word-wrap: break-word;`
- **Truncating text with an ellipsis** by hiding overflow (`overflow: hidden;`), preventing wrapping (`white-space: nowrap;`), and applying `text-overflow: ellipsis;`

Text alignment in tables

Text inside table cells `<td>` and `<th>` can be aligned **horizontally** and **vertically**.

Horizontal alignment: Horizontal alignment is controlled with `text-align` and affects how text is positioned left to right inside a cell.

Value	Effect
`left`	Aligns text to the left
`center`	Centers text horizontally
`right`	Aligns text to the right
`start`	Aligns to the start of the text direction
`end`	Aligns to the end of the text direction

Vertical alignment: Vertical alignment is controlled with `vertical-align` and affects how text is positioned top to bottom inside a cell.

Value	Effect
`top`	Aligns text to the top of the cell
`middle`	Centers text vertically
`bottom`	Aligns text to the bottom of the cell

The `:nth-child()` selector

The `:nth-child()` **pseudo-class** lets you select table rows (or any elements) **based on their position** within their parent element.

Common `:nth-child()` patterns:

Selector	What it selects
`tr:nth-child(1)`	The **first** `<tr>` within its parent
`tr:nth-child(2)`	The **second** `<tr>` within its parent
`tr:nth-child(3)`	The **third** `<tr>` within its parent
`tr:nth-child(even)` or `tr:nth-child(2n)`	Even-numbered rows (2nd, 4th, 6th, …)
`tr:nth-child(odd)` or `tr:nth-child(2n+1)`	Odd-numbered rows (1st, 3rd, 5th, …)
`tr:nth-child(5n+2)`	Rows 2, 7, 12, …

Recap with a practical example

HTML:

```html
<table class="grades">
  <caption>Student Grades</caption>

  <colgroup>
    <col span="1">
    <col span="2" class="name-cols">
    <col span="1" class="score-col">
  </colgroup>

  <thead>
    <tr>
      <th></th>
      <th scope="col">First name</th>
      <th scope="col">Last name</th>
      <th scope="col">Score</th>
    </tr>
  </thead>

  <tbody>
    <tr>
      <th scope="row">Student 1</th>
      <td>Pauline</td>
      <td>Schuermans</td>
      <td>12</td>
    </tr>
    <tr>
      <th scope="row">Student 2</th>
      <td>Tim</td>
      <td>Geyskens</td>
      <td>14</td>
    </tr>
    <tr>
      <th scope="row" rowspan="2">Exchange students</th>
      <td colspan="2">Guest student A</td>
      <td>10</td>
    </tr>
    <tr>
      <td colspan="2">Guest student B</td>
      <td>11</td>
    </tr>
  </tbody>

  <tfoot>
    <tr>
      <th scope="row" colspan="3">Average score</th>
      <td>11.75</td>
    </tr>
  </tfoot>
</table>
```

CSS:

```
table {
  width: 100%;
  border-collapse: collapse;
  table-layout: auto;
}

caption {
  font-weight: bold;
  margin-bottom: 0.5rem;
}

th,
td {
  border: 1px solid #333;
  padding: 0.5rem;
  text-align: left;
}

thead th {
  background-color: #444;
  color: white;
}

.name-cols {
  background-color: #f0f8ff;
}

.score-col {
  background-color: #e6f7ff;
  text-align: center;
}

tbody tr:nth-child(even) {
  background-color: #f9f9f9;
}

tfoot th,
tfoot td {
  font-weight: bold;
  background-color: #eee;
}
```

Positioning

Past vs present page layouts

Historically: tables & floats

Tables were often misused for page layout, even though they are meant for tabular data. This resulted in messy HTML, poor readability, and accessibility issues.

Later floats became popular to place elements side by side. This resulted in:

- parent containers not expanding with floated children
- complex clearing hacks
- layout depended on element order instead of intent

Today: flexbox & grid

Today, we have layout systems designed specifically for page structure: Flexbox and Grid. Most layout logic now lives in CSS, instead of being forced into the HTML structure. This results in:

- flexible and responsive layouts
- clear alignment and spacing
- semantic, readable HTML

Structural elements

Division using `<div>` and `<span>`

A `<div>` is a generic **block-level** container used to group elements together. It has no semantic meaning by itself, but it can be styled with CSS.

A `<span>` is a generic **inline** container used to group part of a text. Like `<div>`, it has no semantic meaning, only styling value.

Remember: `div` elements are useful for grouping content, but **semantic elements should be preferred** whenever possible.

Float

The `float` property allows an element to **float to the left or right** inside its container. Surrounding text and inline elements will **wrap around it**. Today this is commonly used in floating images inside text, such as **articles**.

Using `float` for layout is considered malpractice.

Box-sizing

Default: `box-sizing: content-box`

By default, an element's width and height apply **only to the content**, excluding padding and border. This means **padding and borders are added on top**, making the element larger than expected.

`box-sizing: border-box`

With `box-sizing: border-box`, padding and border are **included inside** the specified width and height.

This is considered **best practice**, developers should add this styling to their wildcard class:

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

Positioning types

Default: static

No positioning logic is applied:

- Follows the normal document flow
- Block elements stack vertically, inline elements flow horizontally
- top, right, bottom, left do nothing

Relative

Moves relative to itself, but still occupies its original space:

- Stays in the normal flow, moves relative to itself
- Can be visually offset using top / left / right / bottom
- Original space is preserved

Absolute

Positioned inside a positioned parent, without affecting layout:

- Removed from the normal flow
- Positioned relative to the **nearest positioned ancestor**
- If none exists, it will be positioned relative to the `<body>` / viewport

Fixed

Fixed to the screen, not the page:

- Removed from the normal flow
- Positioned relative to the viewport
- Does not move when scrolling

Sticky

Relative until it sticks, then behaves like fixed:

- Starts as relative
- Becomes fixed after reaching a scroll threshold
- Requires at least one of: top, left, right, bottom

Z-index

The z-index property controls the **stacking order** of elements that overlap each other. Elements with a higher z-index value appear **in front of elements** with a lower value.

z-index only works on positioned elements:

- `position: relative`
- `position: absolute`
- `position: fixed`
- `position: sticky`

It has no effect on position: `static` elements.

Flexbox

Flexbox is a CSS layout model that controls how the **direct children** of a container are laid out. It is designed for one-dimensional layouts: either a **row** or a **column**.

Any element (often a `div`) becomes a flex container when you set `display: flex`:

- Only the **direct child** elements become flex items
- Nested elements are not affected unless they are also flex containers

Flexbox always works with two **axes**:

- **Main axis** → controlled by `flex-direction`
  - `flex-direction: row;`
  - `flex-direction: column;`
- **Cross axis** → perpendicular to the main axis

Aligning items across the **main axis** using `justify-content`:

- `flex-start`
- `center`
- `flex-end`
- `space-between`
- `space-around`
- `space-evenly`

Aligning items across the **cross axis** using `align-items`:

- `stretch (default)`
- `flex-start`
- `center`
- `flex-end`
- `baseline`

Flex item properties (children)

Each flex item can control how it grows or shrinks.

`flex-grow` dictates how much an item **grows** relative to others:

- 0 = does not grow
- Higher number = grows more

`flex-shrink` dictates how much an item **shrinks** when space is limited:

- Higher number = shrinks more

`flex-basis` dictates the **starting size** before grow/shrink is applied.

We can also choose to override an item itself using the `align-self` property.

Grid

Grid is a layout model that lets you place elements in a two-dimensional grid: **rows** and **columns** at the same time.

Where Flexbox is best for **one direction** (row or column), Grid is designed for **full page and section layouts**.

For more information refer to display documentation.

CSS layout recap

The big picture here is that CSS layout is not one thing or one system. It is several independent systems that answer different questions:

Question	CSS system
How does an element behave by default?	**Display type**: baseline behavior of an element
How are children laid out inside a container?	**Layout model (Flexbox / Grid)**: how children are arranged
How can I move an element from its normal place?	**Positioning**: only affects positioning this element, not it's children.
How do elements overlap?	**z-index**: stacks elements in layers

Responsive design

Responsive design ensures that a website adapts to **different screen sizes** and devices (desktop, tablet, mobile). This is essential because users access websites on many types of screens with varying resolutions.

Responsive design is mainly **achieved using CSS**, not HTML structure.

Key building blocks:

- Flexible **layouts using Flexbox and Grid**
- **Relative units** (`%`, `vw`, `vh`, `rem`) instead of fixed pixels
- **Media queries** to change styles at specific screen widths

Testing Responsiveness with DevTools

DevTools are built-in browser tools used by web developers to inspect, edit, and debug websites.
They are primarily used during front-end development and testing.

DevTools allow you to:

- Inspect HTML and CSS
- Edit styles and layout live
- Debug layout and responsiveness issues

You can open DevTools using:

- Right-click → **Inspect**

- `F12` (mostly Windows/Linux)
- `Ctrl + Shift + I` (Windows)
- `Command + Option + I` (Mac)

Chrome DevTools include a **Device Toolbar** that allows you to:

- Select a device preset (phone, tablet, desktop)
- Enter a custom screen resolution
- Simulate different viewport sizes
- Test responsive layouts without real devices

Viewport

The viewport is the visible area of a web page on a device.
It determines how content is scaled and displayed on different screen sizes.

To control this behavior, a viewport `<meta>` tag is placed inside the `<head>` of an HTML document:

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Without this meta tag:

- **Mobile browsers assume a large desktop width**
- The **page is scaled down**, making text and layout hard to read

Grid Systems

Not to be confusing with the layout type: `display: grid`. This topic is about **grid systems** used to structure page layouts.

Grid systems divide a page into a fixed number of columns
(often **12 columns**), which makes layouts consistent and predictable.

These systems are commonly used in **CSS frameworks** such as `Bootstrap` or `Foundation`.

Older classic grid systems were percentage-based and used hard-coded **width percentages** to define columns. This approach works well but requires manual percentage calculations and is less flexible to maintain.

Example:

- 12 columns → each column is `100% / 12 = 8.3333%`
- A `.col-6` takes up `50%`

- A `.col-12` takes up `100%`

Modern flexbox-based Grid Systems have the same grid concept but can be implemented using **flexbox** without any hard-coded percentages. Instead of defining widths each column gets a **flex value**. That value represents how many "parts" of the available space it takes. This system requires no manual math, is easier to adjust and is more flexible and responsive.

Example:

- `.col-3` → `flex: 3`
- `.col-9` → `flex: 9`

These grid systems are mainly used to create consistent layouts, align content cleanly and build responsive designs efficiently.

By combining a grid system with **media queries**,
you can easily adapt layouts for different screen sizes.

> Modern layouts often use **flexbox or CSS Grid directly**, but grid systems are still useful for understanding layout structure and for working with frameworks.

Media Queries

Media queries are a core tool for **responsive design** and allow you to apply CSS **conditionally**, based on **characteristics of the device or viewport**.

Media query syntax looks like this:

```css
@media screen and (max-width: 600px) {
  body {
    background-color: lightblue;
  }
}
```

Media types:

- `all`: applies to all media types (default)
- `screen`: screens (desktop, tablet, mobile)
- `print`: print or print preview (often used to hide navigation, simplify layout, adjust font sizes)
- `speech`: screen readers

In practice, `screen` and `all` are used most often.

Breakpoint

A breakpoint in CSS is a point where styles change based on a condition.

Breakpoints allow you to:

- adapt layouts
- change font sizes
- stack or unstack columns
- hide or show elements

Examples:

```css
/* Max width */
@media screen and (max-width: 600px) {}

/* Min width */
@media screen and (min-width: 600px) {}

/* Range */
@media screen and (min-width: 600px) and (max-width: 900px) {}
```

Media query external styles & CSS order

The CSS order heavily matters for media queries. Later rules will override earlier ones and that's why media queries are often found at the bottom of stylesheets.

Another common solution is to place a seperate stylesheet after the main one in the `<head>`:

```html
<link rel="stylesheet" media="screen and (min-width: 600px)" href="style.css" />
<link rel="stylesheet" media="screen and (max-width: 600px)" href="style-mobile.css" />
```

Responsive images

Responsive images adapt to the width of their **container**, allowing them to scale correctly on different screen sizes.

This ensures:

- The image scales to the full width of its container
- The aspect ratio is preserved
- The image does not become distorted
- The image doesn't become bigger than a certain amount

```css
img {
  width: 100%;
  height: auto;
  max-width: 600px;
}
```

Forms

A form is an **interactive element** on a web page that allows users to **enter and submit data**. Forms are essential for collecting user input such as login details, search queries, feedback, and more.

Form structure

Elements:

- `<form>`: Required container that defines a form.
- `<input>`: Used to collect user data. Common types include:
  - text
  - number
  - password
  - email
  - checkbox
  - radio
  - submit
- `<label>`: Associates text with an input field, improving accessibility and usability.
- `<button>`: Used for actions. Inside a form, a button **defaults to submitting the form** unless specified otherwise.

`form` attributes:

- `action`: URL where the form data is sent after submission.
- `method`: HTTP method used to send data, usually POST or sometimes GET.
- `enctype`: Defines how form data is **encoded**. For example, `enctype="multipart/form-data"` is required for uploading files.

Submitting a form

A form can be submitted using:

- `<input type="submit">`
- `<button type="submit">`

How form data is sent

When a form is submitted the data is sent as `name-value` pairs where:

- The `name` comes from the name attribute
- The `value` is what the user entered

Inputs without a `name` attribute **are not sent**.

Example:

```html
<!-- Don't worry about the GET method or the URL for now -->
<form action="/submit" method="GET">
  <label for="name">Name</label>
  <input type="text" id="name" name="username" required>

  <label for="email">Email (optional)</label>
  <input type="email" id="email" name="emailAddress">

  <label for="no-name">Input without name</label>
  <input type="text" id="no-name">

  <button>Submit</button>
</form>
```

The URL would look something like: `https://example.com/forms/submit.html?name=TestUser&emailAddr`

The use of buttons within forms

There are three common button types in forms:

- `submit`: Sends the form data to the URL defined in the `action` attribute.
- `reset`: Resets all form fields to their **initial** values.
- `button`: Has **no default behavior** (it does not submit or reset the form) and is typically used for custom actions handled with JavaScript.

These behaviors apply to both `<input>` and `<button>` elements.

`<input>` vs. `<button>`

By default a `<button>` inside a form, **without a `type` attribute** defaults to the `type="submit"` attribute. Best practice is to **always specify the type** depending on what your needs.

`<input>` buttons:

- Button text is defined using the `value` attribute
- Cannot contain other HTML elements

Example:

```html
<input type="submit" value="Submit">
<input type="reset" value="Reset">
<input type="button" value="Click me">
```

`<button>` buttons:

- Button text goes between the tags
- Can contain other HTML elements (icons, spans, images, …)
- More flexible and generally preferred in modern HTML

Example:

```html
<button type="submit">Submit</button>
<button type="reset">Reset</button>
<button type="button">Click me</button>
```

Input types in HTML forms

HTML forms use different `<input>` types to collect user data. Each type affects how data is entered, validated, and sent to the server.

`<input type="text">`

Input fields are elements that allow users to enter data. This data is sent to another page or processed by a server when the form is submitted.

Common attributes:

- `id`: Unique identifier, often used with `<label>`
- `name`: Key used when submitting form data

- `value`: Sets a default value for the input field
- `placeholder`: Temporary hint text that describes the expected input
- `size`: Controls the visual width of the input (in characters)
- `required`: Makes the field mandatory before submitting the form
- `disabled`: Disables the field so the user cannot interact with it
- `readonly`: Makes the field read-only; the value can be selected but not changed
- `minlength` / `maxlength`: Sets character limits
- `autocomplete`: Enables or disables browser autofill suggestions
- `autofocus`: Automatically focuses the field when the page loads
- `pattern`: Regular expression used to validate the input value
- `title`: Tooltip text shown on hover
- `spellcheck`: Enables or disables spell checking
- `tabindex`: Controls keyboard tab navigation order

Example:

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

The `name` attribute determines which data is sent when the form is submitted. In key-value pairs the:

- `name` is the key
- user input is the `value`

For the sake of repetition, if an input does not have a `name` attribute, its value will **not be sent**.

`<input type="password">`

Used for entering sensitive text such as passwords. The input is **visually masked**, but the value itself is **not encrypted**:

- Masking is only visual
- Data is still sent as plain text
- Always use HTTPS
- Never use GET for passwords → use POST

It is the your responsibility to make sure passwords are properly encrypted. Encryption is a large complex topic, therefore it's beyond the scope of this course.

`<input type="email">`

Used for entering e-mail addresses:

- Browser automatically **validates basic e-mail** format
- Requires an `@` and a valid **domain**

You could however customize validation The `pattern` attribute allows stricter validation using a **regular expressions**:

- Can **enforce structure** (e.g. must contain `.`)
- Can **restrict allowed domains** (e.g. only `example.com`)

`<input type="number">`

Used for numeric input only. The browser will block any non-numeric input and validates the value automatically. Commonly used with the `min` and `max` attributes to set a range. The `step` attribute is also popular to set a custom increment size.

`<input type="range">`

A numeric input shown as a **slider**. This is commonly used for values within a fixed range (volume, brightness, etc.).

`<input type="checkbox">`

Used to toggle an option **on** or **off**.

- A checkbox is checked when the `checked` attribute is present
- If unchecked, **no value is sent**

There's an edge case where multiple checkboxes can share the same name. In this case each checked option is sent separately.

Checkboxes are commonly used for **GDPR** reasons like agreeing to terms and conditions, newletters, cookies, etc.

`<input type="radio">`

Used to select **one option from a group**:

- Radio buttons are **grouped by the same `name`**
- Only one option can be selected at a time
- One option in the group **must be selected before submission**

`<input type="date">`

Used for selecting a date:

- Browser provides a **date picker UI**
- Appearance depends on browser and OS
- Can be limited with `min` and `max`

`<input type="hidden">`

An **invisible** input field that **still sends data**:

- Not visible or editable by the user
- Commonly used for **IDs**, **tokens**, or **server-generated values**
- **Included when the form is submitted**

Other input types

There's a whole list of input types available.

`<textarea>`

A `<textarea>` element is used for entering **multi-line** text, such as comments, messages, or descriptions. It is suited for longer input where a single-line `<input>` is insufficient.

Although `rows` and `cols` control the default size, it is common practice to use CSS for `width` and `height` instead. This provides more control and consistent layout across devices.

Common attributes:

- `rows`: The number of visible text lines
- `cols`: The number of visible characters per line
- `maxlength`: The maximum number of characters
- `required`: The field must be filled before submission
- `placeholder`: The hint text shown when empty
- `name`: The key used when sending form data
- `id`: Used to link the textarea to a `<label>`

Unlike `<input>`, the value **is placed** between the opening and closing tags:

```
<textarea>Initial text</textarea>
```

`<select>` and `<option>`

A `<select>` element creates a **dropdown menu** that allows the user to choose one or more options from a predefined list. The available choices are defined using `<option>` elements inside of the `<select>` opening and closing tags.

Common `<select>` attributes:

- `name`: The key used when submitting form data
- `id`: Links the select to a `<label>`
- `multiple`: Allows selecting multiple options
- `size`: The number of visible options without opening the dropdown

Common `<option>` attributes:

- `value`: The value sent to the server when selected
- `selected`: Makes the option selected by default
- `disabled`: Makes it so the option cannot be selected

Example with common placeholder pattern:

```
<select id="single" name="select-single">
  <option value="" selected="" disabled="">Select an option</option>
  <option value="a">A</option>
  <option value="b">B</option>
  <option value="c">C</option>
</select>
```

`<fieldset>` and `<legend>`

The `<fieldset>` element is a **semantic** element used to group **related form controls** together. It improves form structure, readability, and accessibility. A `<legend>` provides a title or description for the group.

Example:

```
<fieldset>
  <legend>Dropdown menu</legend>
  <select id="single" name="select-single">
    <option value="" selected="" disabled="">Select an option</option>
    <option value="a">A</option>
    <option value="b">B</option>
    <option value="c">C</option>
  </select>
</fieldset>
```

`<datalist>`

The `<datalist>` element provides a list of suggested values for an `<input>` field. It enhances user experience by **showing autocomplete suggestions while still allowing free input**.

The `<datalist>` offers more freedom **as opposed to** the `<select>` element, which restricts the user to fixed options.
Best practice is to use a `<datalist>` when you want to assist the user without enforcing strict input. Use `<select>` when the input must be limited to predefined values.

It works by linking a `<datalist>` to an `<input>` element using the `list` attribute.

Example:

```
<input list="example-list" name="example">

<datalist id="example-list">
  <option value="Option A">
  <option value="Option B">
  <option value="Option C">
</datalist>
```

Validation

Validation ensures that user input meets certain rules before it is processed.

So far, we have only seen **frontend validation**. This means the browser checks the user's input **before** the form is submitted, using attributes such as `required`, `pattern`, `min`, `max`, etc.

Frontend validation improves user experience, but it is **not secure**. A user can disable or manipulate browser validation by changing or removing HTML attributes.

For this reason, **backend validation is always required**. The server must validate all incoming data again before storing or processing it.

Long story short: **never** trust user input.

The `POST` and `GET` methods within forms

When a form is submitted, the browser sends the data to the server using an **HTTP** method. The two most common methods are `GET` and `POST` but there's many others like: `PUT`, `DELETE`, `PATCH`, etc.

`GET`:

- Sends form data as query parameters **in the URL**
- Data is **visible** in the address bar
- Values can be stored in browser history and logs
- Suitable for **retrieving** data (e.g. search forms)
- Not safe for sensitive information (passwords, personal data)

Example result: `/submit?name=John+Doe`

`POST`:

- Sends form data in the **request body**
- Data is **not visible** in the URL, this doesn't mean it's secure
- Better suited for **sensitive or larger data**
- Commonly used for creating or updating data

You can inspect `POST` data in the **DevTools network tab**, where the payload appears in the request body instead of the URL.

To truly protect sensitive information, **HTTPS** is required.