Client-Side Web Technologies – CSS

Databases

Pierre Senellart

PSL UNIVERSITÉ PARIS

18 May 2026

Outline

Where we are

- HTML gives us structure and content: tags describe *what* a region *is*
- Browsers apply a generic user-agent stylesheet so that, by default, an `<h1>` looks bigger than a `<p>`
- But every page then looks the same and nothing matches a site's identity

Where we are

- HTML gives us structure and content: tags describe *what* a region *is*
- Browsers apply a generic user-agent stylesheet so that, by default, an `<h1>` looks bigger than a `<p>`
- But every page then looks the same and nothing matches a site's identity

We need a way to control appearance without polluting the structure.

CSS: Cascading Style Sheets

- Open standard maintained by the W3C
- Describes how elements should be rendered: colors, fonts, spacing, layout, responsiveness...
- Kept separate from HTML: same page, many possible stylings (think of dark mode, print mode, mobile vs desktop)
- Cascading: several rules may target the same element; a precise procedure decides who wins
- Reference:
  - `https://www.w3.org/Style/CSS/`
  - `https://developer.mozilla.org/en-US/docs/Web/CSS`

Separation of concerns

HTML	CSS	JavaScript
structure & content	style & layout	behavior

one document, three languages

Separation of concerns

one document, three languages

Each language can be edited independently:

- a designer can restyle the whole site without touching the HTML
- the server can ship the same HTML to every device, and different CSS to phones vs desktops

Outline

Three ways to attach CSS to a page

1. External stylesheet (recommended):

HTML

```
<link rel="stylesheet" href="/static/style.css">
```

Three ways to attach CSS to a page

1. External stylesheet (recommended):

HTML

```
<link rel="stylesheet" href="/static/style.css">
```

2. Internal stylesheet, inside the document:

HTML

```
<style>
  h1 { color: navy; }
</style>
```

Three ways to attach CSS to a page

1. External stylesheet (recommended):

HTML

```
<link rel="stylesheet" href="/static/style.css">
```

2. Internal stylesheet, inside the document:

HTML

```
<style>
  h1 { color: navy; }
</style>
```

3. Inline style, on a single element:

HTML

```
<p style="color: red;">Important!</p>
```

Which to use?

external default choice: one file, cacheable, reusable, easy to swap

internal handy for a quick experiment or a single-page document

inline avoid in general: defeats separation of concerns, hard to maintain, but unavoidable when styles come from data (e.g. a per-row color computed in Python)

Outline

Rule syntax

A stylesheet is a list of rules, each of the form:

CSS

```
selector {
  property: value;
  property: value;
}
```

Example:

CSS

```
h1 {
  color: navy;
  font-size: 2em;
}
```

selector picks which elements the rule applies to

declarations `property: value;` pairs, in any order; the final `;` is optional but conventional

Comments and whitespace

Comments only have one form:

CSS

```css
/* this is a CSS comment;
   it can span several lines */
```

Whitespace between tokens is mostly free; the snippets

CSS

```css
h1{color:navy;font-size:2em;}
```

and

CSS

```css
h1 {
  color: navy;
  font-size: 2em;
}
```

are equivalent. Use whichever is readable.

Outline

Basic selectors

type `h1`, `p`, `table`... every element with that tag

class `.warning`: every element whose `class` attribute contains `warning`

id `#contact`: the unique element whose `id` attribute is `contact`

universal `*`: every element (use sparingly)

attribute `a[target="_blank"]`: every `<a>` whose `target` attribute is `_blank`

Basic selectors

type `h1`, `p`, `table`… every element with that tag

class `.warning`: every element whose `class` attribute contains `warning`

id `#contact`: the unique element whose `id` attribute is `contact`

universal `*`: every element (use sparingly)

attribute `a[target="_blank"]`: every `<a>` whose `target` attribute is `_blank`

Convention: use a class when several elements share the styling, an id for a single named element.

Classes in HTML

Attach a class in the HTML:

HTML

```
<p class="warning">Booking fully refundable until 24h before
↪  arrival.</p>
<p class="warning highlighted">Sold out for this weekend.</p>
```

Style it in the CSS:

CSS

```
.warning      { color: darkred; }
.highlighted  { background: yellow; }
```

Classes in HTML

Attach a class in the HTML:

HTML

```
<p class="warning">Booking fully refundable until 24h before
↪  arrival.</p>
<p class="warning highlighted">Sold out for this weekend.</p>
```

Style it in the CSS:

CSS

```
.warning      { color: darkred; }
.highlighted  { background: yellow; }
```

An element can carry several classes (space-separated): both rules will apply to the second paragraph.

Combinators

Pick elements by their position in the DOM tree:

- `A B` descendant: any `B` inside an `A`
- `A > B` child: `B` directly inside `A`
- `A + B` adjacent sibling: `B` right after `A`
- `A, B` grouping: same declarations for both `A` and `B`

Example: alternate row colors only in the body of a table.

CSS

```
table tbody tr { background: white; }
table tbody tr:nth-child(even) { background: #f0f0f0; }
```

Pseudo-classes

Pseudo-classes target elements in a particular state or position, with a leading colon:

`:hover` mouse is over the element

`:focus` element has keyboard focus (forms, accessibility)

`:visited` visited link

`:first-child`, `:last-child` first/last child of its parent

`:nth-child(n)` nth child; `:nth-child(even)` useful for tables

`:not(s)` anything not matching the selector `s`

Pseudo-classes

Pseudo-classes target elements in a particular state or position, with a leading colon:

`:hover` mouse is over the element

`:focus` element has keyboard focus (forms, accessibility)

`:visited` visited link

`:first-child`, `:last-child` first/last child of its parent

`:nth-child(n)` nth child; `:nth-child(even)` useful for tables

`:not(s)` anything not matching the selector `s`

Example:

CSS

```
a:hover  { text-decoration: underline; }
input:focus { outline: 2px solid navy; }
```

Pseudo-elements

Pseudo-elements (two colons) refer to a part of an element that has no tag of its own:

`::first-line`, `::first-letter` typography effects

`::before`, `::after` insert generated content before/after the element's content

`::placeholder` the placeholder text of an input field

Pseudo-elements

Pseudo-elements (two colons) refer to a part of an element that has no tag of its own:

`::first-line`, `::first-letter` typography effects

`::before`, `::after` insert generated content before/after the element's content

`::placeholder` the placeholder text of an input field

Example: mark required form fields automatically.

CSS

```css
label.required::after {
  content: " *";
  color: red;
}
```

Outline

When several rules match

Two rules with conflicting declarations may target the same element. Who wins? Three mechanisms, applied in order:

1. origin and importance: author-provided styles beat browser defaults; `!important` declarations beat everything else (use sparingly)
2. specificity: a measure of how precisely the selector targets the element
3. source order: on ties, the rule appearing later in the stylesheet wins

When several rules match

Two rules with conflicting declarations may target the same element. Who wins? Three mechanisms, applied in order:

1. origin and importance: author-provided styles beat browser defaults; `!important` declarations beat everything else (use sparingly)
2. specificity: a measure of how precisely the selector targets the element
3. source order: on ties, the rule appearing later in the stylesheet wins

Example: `!important` overrides a more specific rule.

CSS

```
#submit { background: blue; }                /* high specificity */
button  { background: red !important; }      /* wins anyway */
```

Specificity in a nutshell

Count, for each selector, a triple (a, b, c):

- a number of id selectors (`#contact`)
- b number of class, attribute, pseudo-class selectors (`.warning`, `[type=text]`, `:hover`)
- c number of type and pseudo-element selectors (`h1`, `::before`)

Triples are compared lexicographically, a first.

Specificity in a nutshell

Count, for each selector, a triple (a, b, c):

- a number of id selectors (`#contact`)
- b number of class, attribute, pseudo-class selectors (`.warning`, `[type=text]`, `:hover`)
- c number of type and pseudo-element selectors (`h1`, `::before`)

Triples are compared lexicographically, a first. Example:

CSS

```
p                  { color: black;  }   /* (0,0,1) */
.warning           { color: darkred;}   /* (0,1,0) -> wins over p */
#contact p.warning { color: orange; }   /* (1,1,1) -> wins above */
```

Inheritance

- Some properties are automatically inherited by descendants: `color`, `font-family`, `font-size`, `line-height`...
- Others are not: `margin`, `padding`, `border`, `background`... (they would not make sense)
- Set a property on the `<body>` and the whole page picks it up

Inheritance

- Some properties are automatically inherited by descendants: `color`, `font-family`, `font-size`, `line-height`...
- Others are not: `margin`, `padding`, `border`, `background`... (they would not make sense)
- Set a property on the `<body>` and the whole page picks it up

Practical consequence: configure typography once on `body`; override only where needed.

Outline

Colors

Many equivalent ways to write the same color:

CSS

```css
color: red;                  /* named (147 keywords) */
color: #ff0000;              /* 6-hex RGB */
color: #f00;                 /* 3-hex shorthand */
color: rgb(255, 0, 0);       /* decimal RGB */
color: rgb(255 0 0 / 50%); /* with alpha (transparency) */
color: hsl(0, 100%, 50%);  /* hue, saturation, lightness */
```

Colors

Many equivalent ways to write the same color:

CSS

```css
color: red;                  /* named (147 keywords) */
color: #ff0000;              /* 6-hex RGB */
color: #f00;                 /* 3-hex shorthand */
color: rgb(255, 0, 0);       /* decimal RGB */
color: rgb(255 0 0 / 50%); /* with alpha (transparency) */
color: hsl(0, 100%, 50%);  /* hue, saturation, lightness */
```

HSL is the most human-friendly: pick a hue (0–360), then tune saturation and lightness.

Sizes: absolute and relative units

`px` CSS pixel; consistent across devices (a CSS pixel is *not* a physical pixel)

`pt` point (1 pt = 1/72 inch); legacy, mostly used in print stylesheets

`em` relative to the parent's font size

`rem` relative to the root's font size (`<html>`); preferred for predictable scaling

`ch` width of the digit "0" – handy for sizing text columns

`%` percentage of the parent's corresponding value

`vw`, `vh` 1% of the viewport width / height

Sizes: absolute and relative units

`px`	CSS pixel; consistent across devices (a CSS pixel is *not* a physical pixel)
`pt`	point ($1\,\text{pt} = 1/72\,\text{inch}$); legacy, mostly used in print stylesheets
`em`	relative to the parent's font size
`rem`	relative to the root's font size (`<html>`); preferred for predictable scaling
`ch`	width of the digit "0" – handy for sizing text columns
`%`	percentage of the parent's corresponding value
`vw`, `vh`	1% of the viewport width / height

Accessibility: the user can change the default font size in the browser. Absolute units (`px`, `pt`) ignore this preference; relative units (`rem`, `em`) honor it.

Outline

Font families

Five generic families, available everywhere:

`serif`	small strokes at the end of letters; long-form reading (Times, Garamond…)
`sans-serif`	no serifs; default for screens (Helvetica, Arial…)
`monospace`	every glyph has the same width; code, tabular data (`Courier`, `Menlo`…)
`cursive`	handwriting style (*Zapf Chancery*, *Brush Script*…); use sparingly
`system-ui`	the operating system's UI font

Font families

Five generic families, available everywhere:

`serif`	small strokes at the end of letters; long-form reading (Times, Garamond…)
`sans-serif`	no serifs; default for screens (Helvetica, Arial…)
`monospace`	every glyph has the same width; code, tabular data (`Courier`, `Menlo`…)
`cursive`	handwriting style (*Zapf Chancery*, *Brush Script*…); use sparingly
`system-ui`	the operating system's UI font

In practice, declare a font stack: preferred name first, generic family as a last-resort fallback.

CSS

```
body { font-family: "Inter", "Helvetica Neue", Arial, sans-serif; }
code { font-family: "Fira Code", Menlo, Consolas, monospace; }
```

Font properties

`font-size` size of the glyphs; usually in `rem` or `em`

`font-weight` thickness, from `100` (thin) to `900` (**black**); keywords `normal` (≈ 400) and `bold` (≈ 700)

`font-style` `normal` or *`italic`*

`font-variant` e.g. `small-caps`

`font` shorthand for all the above plus `line-height` and `font-family`

Font properties

`font-size` size of the glyphs; usually in `rem` or `em`

`font-weight` thickness, from `100` (thin) to `900` (**black**); keywords `normal` (≈ 400) and `bold` (≈ 700)

`font-style` `normal` or *`italic`*

`font-variant` e.g. `small-caps`

`font` shorthand for all the above plus `line-height` and `font-family`

Use the semantic HTML tags (`<strong>`, `<em>`...) and let CSS decide how they look. Do not pick `<b>` only because you want bold somewhere.

Spacing and alignment

`line-height` vertical space between lines (the typographer's leading); a unitless value like `1.5` scales with `font-size`

`letter-spacing`, `word-spacing` horizontal spacing; negative values bring characters closer (use sparingly)

`text-align`: `left`, `right`, `center`, `justify`

`text-decoration`: `underline`, `line-through`...

`text-transform`: `uppercase`, `lowercase`, `capitalize`

Spacing and alignment

`line-height` vertical space between lines (the typographer's leading); a unitless value like `1.5` scales with `font-size`

`letter-spacing`, `word-spacing` horizontal spacing; negative values bring characters closer (use sparingly)

`text-align`: `left`, `right`, `center`, `justify`

`text-decoration`: `underline`, `line-through`...

`text-transform`: `uppercase`, `lowercase`, `capitalize`

Sensible defaults for the body of a page:

CSS

```
body { font-size: 1rem; line-height: 1.5; max-width: 65ch; }
```

Typography and accessibility

Reading on screen is harder than on paper. A few habits help every reader, and accessibility laws expect them:

- Keep body text at least at the user's default size (`1rem`); never go below $\approx$ 12 px
- Aim for $\approx$ 60–75 characters per line: shorter is easier to follow than the full width of the screen
- Generous line-height (1.4–1.6) makes paragraphs scannable
- Ensure enough color contrast between text and background (the WCAG guideline is a contrast ratio of $\geqslant 4.5$ for normal text)
- Do not communicate information by color alone: also vary text, icons or position
- Keep focus outlines visible: `outline: none` strands keyboard users

Beyond system fonts

To use a font that may not exist on the user's machine, either link a hosted stylesheet:

HTML

```
<link href="https://fonts.googleapis.com/css2?family=Inter"
      rel="stylesheet">
```

or self-host the file and declare it:

CSS

```
@font-face {
  font-family: "Inter";
  src: url("/static/Inter.woff2") format("woff2");
  font-display: swap;
}
```

Beyond system fonts

To use a font that may not exist on the user's machine, either link a hosted stylesheet:

HTML

```
<link href="https://fonts.googleapis.com/css2?family=Inter"
      rel="stylesheet">
```

or self-host the file and declare it:

CSS

```
@font-face {
  font-family: "Inter";
  src: url("/static/Inter.woff2") format("woff2");
  font-display: swap;
}
```

Each extra font is an extra HTTP request and slows the page; one or two well-chosen families is plenty.

Hosted fonts: a privacy and GDPR trap

Linking a stylesheet from `fonts.googleapis.com` (or any third-party CDN) looks innocuous, but every visit to your page triggers an HTTP request that hands Google:

- the user's IP address
- the page's URL (via the `Referer` header)
- a browser fingerprint (`User-Agent...`)

Hosted fonts: a privacy and GDPR trap

Linking a stylesheet from `fonts.googleapis.com` (or any third-party CDN) looks innocuous, but every visit to your page triggers an HTTP request that hands Google:

- the user's IP address
- the page's URL (via the `Referer` header)
- a browser fingerprint (`User-Agent...`)

Under the GDPR (EU) this is the processing of personal data by a third party, generally without the consent the law requires:

- a German court (Munich, 2022) explicitly ruled that embedding Google Fonts is unlawful without consent
- the Austrian and French data-protection authorities have ruled on the closely related case of Google Analytics, with the same reasoning
- the logic applies to any third-party analytics, ads, maps, video embeds...

Hosted fonts: a privacy and GDPR trap

Linking a stylesheet from `fonts.googleapis.com` (or any third-party CDN) looks innocuous, but every visit to your page triggers an HTTP request that hands Google:

- the user's IP address
- the page's URL (via the `Referer` header)
- a browser fingerprint (`User-Agent...`)

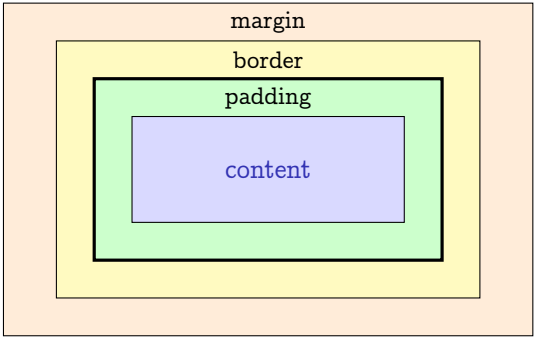
Under the GDPR (EU) this is the processing of personal data by a third party, generally without the consent the law requires:

- a German court (Munich, 2022) explicitly ruled that embedding Google Fonts is unlawful without consent
- the Austrian and French data-protection authorities have ruled on the closely related case of Google Analytics, with the same reasoning
- the logic applies to any third-party analytics, ads, maps, video embeds...

Safer default: self-host the font files, or rely on the user's system fonts. Treat every external resource as a privacy decision.

Outline

Every element is a box



content	the actual text / image / child boxes
padding	transparent space inside the border
border	visible frame
margin	transparent space outside the border, separating the box from its neighbors

Setting box properties

CSS

```css
.card {
  padding: 1rem;              /* same on all four sides */
  border: 1px solid #888;
  margin: 2rem auto;          /* 2rem top/bottom, auto left/right */
  background: #fafafa;
}
```

Individual sides can be set:

CSS

```css
.card {
  padding-top: 0.5rem;
  padding-bottom: 1.5rem;
  margin: 1rem 2rem 0 2rem;  /* top right bottom left */
}
```

Setting box properties

CSS

```css
.card {
  padding: 1rem;              /* same on all four sides */
  border: 1px solid #888;
  margin: 2rem auto;          /* 2rem top/bottom, auto left/right */
  background: #fafafa;
}
```

Individual sides can be set:

CSS

```css
.card {
  padding-top: 0.5rem;
  padding-bottom: 1.5rem;
  margin: 1rem 2rem 0 2rem;  /* top right bottom left */
}
```

The width trap: box-sizing

By default, `width` sets the content width; padding and border are added on top:

CSS

```css
.card {
  width: 300px;          /* only the content area */
  padding: 20px;
  border: 5px solid;     /* total rendered width: 350px */
}
```

The width trap: box-sizing

By default, `width` sets the content width; padding and border are added on top:

CSS

```css
.card {
  width: 300px;          /* only the content area */
  padding: 20px;
  border: 5px solid;     /* total rendered width: 350px */
}
```

Almost every modern stylesheet starts with:

CSS

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

Now `width` is the total width (content + padding + border), which is what most people expect.

`display`: changing the box's nature

Recall that HTML elements are block or inline by default. CSS can change this:

`display: block` full width, starts on a new line

`display: inline` only as wide as the content, in flow

`display: inline-block` flows like inline, but accepts width / height / vertical padding

`display: none` removed from the layout (and from accessibility tools) entirely

`display: flex`, `display: grid` turn the element into a layout container (next sections)

`display`: changing the box's nature

Recall that HTML elements are block or inline by default. CSS can change this:

`display: block` full width, starts on a new line

`display: inline` only as wide as the content, in flow

`display: inline-block` flows like inline, but accepts width / height / vertical padding

`display: none` removed from the layout (and from accessibility tools) entirely

`display: flex`, `display: grid` turn the element into a layout container (next sections)

Note: `visibility: hidden` hides the element but keeps its space; `display: none` removes the space too.

Outline

The problem with default flow

By default, HTML lays out:

- block elements stacked vertically
- inline elements flowing horizontally, wrapping at the right edge

This is not enough as soon as we want:

- a navigation bar with items spread evenly
- a card layout with rows of equal-height boxes
- a footer that sticks to the bottom of the viewport

The problem with default flow

By default, HTML lays out:

- block elements stacked vertically
- inline elements flowing horizontally, wrapping at the right edge

This is not enough as soon as we want:

- a navigation bar with items spread evenly
- a card layout with rows of equal-height boxes
- a footer that sticks to the bottom of the viewport

Two modern tools: Flexbox (one-dimensional, this section) and Grid (two-dimensional, next section).

Flexbox: a one-dimensional layout

Declare a container as a flex container:

CSS

```css
.toolbar {
  display: flex;
  gap: 1rem;
}
```

All its direct children become flex items laid out along one axis (row by default).

Flexbox: a one-dimensional layout

Declare a container as a flex container:

CSS

```css
.toolbar {
  display: flex;
  gap: 1rem;
}
```

All its direct children become flex items laid out along one axis (row by default).

Aligning flex items

`flex-direction` `row` (default) or `column`; reverse variants exist

`justify-content` alignment along the main axis: `flex-start`, `center`, `flex-end`, `space-between`, `space-around`

`align-items` alignment along the cross axis: `stretch` (default), `center`, `flex-start`, `flex-end`

`gap` space between items (replaces nasty margin tricks)

`flex-wrap` `wrap` to allow several lines

Aligning flex items

`flex-direction` `row` (default) or `column`; reverse variants exist

`justify-content` alignment along the main axis: `flex-start`, `center`, `flex-end`, `space-between`, `space-around`

`align-items` alignment along the cross axis: `stretch` (default), `center`, `flex-start`, `flex-end`

`gap` space between items (replaces nasty margin tricks)

`flex-wrap` `wrap` to allow several lines

CSS

```css
.navbar {
  display: flex;
  justify-content: space-between;
  align-items: center;
  gap: 1rem;
}
```

Items that grow

Each flex item can be told how to share leftover space:

CSS

```css
.sidebar { flex: 0 0 200px; }   /* fixed 200px */
.main    { flex: 1; }           /* take all remaining space */
```

Items that grow

Each flex item can be told how to share leftover space:

CSS

```css
.sidebar { flex: 0 0 200px; }   /* fixed 200px */
.main    { flex: 1; }           /* take all remaining space */
```

The `flex` shorthand stands for

`flex-grow` `flex-shrink` `flex-basis`

`flex-grow` how greedy the item is for extra space

`flex-shrink` how willing it is to give up space when cramped

`flex-basis` its preferred size before grow/shrink

Outline

CSS Grid: a two-dimensional layout

Flexbox handles one axis; Grid handles rows *and* columns at once:

CSS

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

Three equal-width columns; rows are created automatically as items are added.

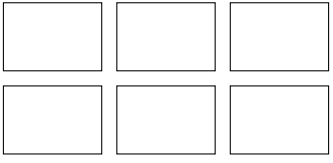
CSS Grid: a two-dimensional layout

Flexbox handles one axis; Grid handles rows *and* columns at once:

CSS

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

Three equal-width columns; rows are created automatically as items are added.



Track sizes and the `fr` unit

`1fr` fractional unit: shares of the available space after fixed sizes have been allocated

`minmax(min, max)` grow within bounds

`auto-fit`, `auto-fill` create as many tracks as fit

A responsive card grid in one line:

CSS

```css
.cards {
  display: grid;
  grid-template-columns:
    repeat(auto-fit, minmax(15rem, 1fr));
  gap: 1rem;
}
```

Track sizes and the `fr` unit

`1fr` fractional unit: shares of the available space after fixed sizes have been allocated

`minmax(min, max)` grow within bounds

`auto-fit`, `auto-fill` create as many tracks as fit

A responsive card grid in one line:

CSS

```css
.cards {
  display: grid;
  grid-template-columns:
    repeat(auto-fit, minmax(15rem, 1fr));
  gap: 1rem;
}
```

Effect: as many 15 rem-wide columns as fit the viewport; leftover space is shared equally.

Flexbox or Grid?

Flexbox for one-dimensional arrangements: a navbar, a toolbar, a row of buttons, a footer

Grid for two-dimensional arrangements: a card gallery, a dashboard, a page layout with header / sidebar / main / footer

Flexbox or Grid?

Flexbox for one-dimensional arrangements: a navbar, a toolbar, a row of buttons, a footer

Grid for two-dimensional arrangements: a card gallery, a dashboard, a page layout with header / sidebar / main / footer

In practice, the two are often combined: a Grid divides the page into regions, and inside each region a Flex container places the items.

Outline

The viewport meta tag

Mobile browsers pretend the screen is 980 px wide and zoom out, unless told otherwise. Add to the <head>:

HTML

```
<meta name="viewport"
      content="width=device-width, initial-scale=1">
```

Now the browser uses the actual screen width, and CSS pixels match the device.

Media queries

Apply a block of CSS only conditionally:

CSS

```css
.layout {
  display: grid;
  grid-template-columns: 1fr;        /* phone: one column */
}

@media (min-width: 50rem) {
  .layout {
    grid-template-columns: 1fr 2fr; /* tablet+: two columns */
  }
}
```

Media queries

Apply a block of CSS only conditionally:

CSS

```css
.layout {
  display: grid;
  grid-template-columns: 1fr;        /* phone: one column */
}

@media (min-width: 50rem) {
  .layout {
    grid-template-columns: 1fr 2fr; /* tablet+: two columns */
  }
}
```

Common conditions: `min-width`, `max-width`, `orientation: landscape`, `prefers-color-scheme: dark`.

Mobile-first

Two ways to write a responsive stylesheet:

mobile-first start with the styles for the smallest screen; layer extra rules behind `@media (min-width: ...)` as the viewport grows

desktop-first start with the styles for the largest screen and shrink with `@media (max-width: ...)`

Mobile-first

Two ways to write a responsive stylesheet:

mobile-first start with the styles for the smallest screen; layer extra rules behind `@media (min-width: ...)` as the viewport grows

desktop-first start with the styles for the largest screen and shrink with `@media (max-width: ...)`

Mobile-first is now the recommended default: most traffic is on phones, and adding things is easier than overriding.

Outline

A readable reservations table

Recall that an SQL query result becomes an HTML `<table>`. A small stylesheet makes it readable:

CSS

```css
table { border-collapse: collapse; width: 100%; }
th, td {
  text-align: left;
  padding: 0.4rem 0.8rem;
  border-bottom: 1px solid #ddd;
}
thead { background: #f0f0f0; }
tbody tr:nth-child(even) { background: #fafafa; }
tbody tr:hover           { background: #ffffd0; }
```

A readable reservations table

Recall that an SQL query result becomes an HTML `<table>`. A small stylesheet makes it readable:

CSS

```
table { border-collapse: collapse; width: 100%; }
th, td {
  text-align: left;
  padding: 0.4rem 0.8rem;
  border-bottom: 1px solid #ddd;
}
thead { background: #f0f0f0; }
tbody tr:nth-child(even) { background: #fafafa; }
tbody tr:hover           { background: #ffffd0; }
```

Zebra stripes, a header band and a hover effect, in seven rules. Tabular data is the natural rendering of a relation, and CSS makes it pleasant.

A booking form, laid out

CSS

```
form          { display: grid; gap: 0.6rem 1rem;
                grid-template-columns: max-content 1fr;
                max-width: 30rem; }
form label    { grid-column: 1; align-self: center; }
form input,
form select   { grid-column: 2; }
form button   { grid-column: 2; justify-self: start;
                padding: 0.4rem 1rem; }
```

Grid aligns the labels in one column and the fields in another, with no table markup.

Outline

What CSS gives us

- Total control over appearance, kept separate from the document's structure and content
- A small set of primitives (selectors, the box model, Flexbox/Grid, media queries) that combine into every layout you see on the Web
- Cascading and inheritance let a handful of base rules style a whole site, with targeted overrides where needed
- A direct path from SQL query result to a usable, legible, responsive Web page, without touching the HTML the server emits

What CSS does *not* do

- Add or remove content based on data: that is HTML generated by templates on the server
- React to clicks, type-ahead, dynamic updates: that is JavaScript
- Talk to the database: that is the server's job

To explore on your own

- MDN Web Docs, the Web reference:
  `https://developer.mozilla.org/en-US/docs/Web/CSS`
- A Complete Guide to Flexbox:
  `https://css-tricks.com/snippets/css/a-guide-to-flexbox/`
- A Complete Guide to Grid:
  `https://css-tricks.com/snippets/css/complete-guide-grid/`
- In the browser, the developer tools (F12): inspect any element, see which rules apply, and edit them live
- Pure-CSS galleries to read: `https://csszengarden.com/`

License