CSS Overview

- Rules for appearance of **rendered** HTML
- Based on structure of HTML

Usually separate from HTML

- Can be used with multiple HTML documents
    - Where structure matches

Stylesheets

There are a few ways to apply CSS to HTML

- On an element using `style` attribute (**don't do**)
- In document `<style>` element in `<head>` (**don't do**)
- A stylesheet file linked via `<link>` element

Option 1: Use a `style` attribute (Don't do)

Apply "inline css" to an element using a **style attribute**

```
<div style="color: red;">Example</div>
```

Generally: Don't use `style` attribute!

- A few exceptions exist
- For this course: **DO NOT USE `style` ATTRIBUTE**!

Why not use `style` attribute?

- Hard to override
- Impossible to reuse CSS
- Really annoying to edit
- Frustrating to debug
- Difficult to maintain

> *This attribute and the `<style>` element have mainly the purpose of allowing for quick styling, for example for testing purposes -- MDN*

Option 2: Using a `<style>` element (Don't do)

Define rules in `<style>` to apply to matching elements

```
<head>
  <style>
    #demo {
      color: red;
    }

    .selected {
      background-color: red;
    }
  </style>
</head>
<body>
  <div id="demo">Example</div>
</body>
```

- Generally: Don't do this
- For this course: **Do not *write* this**

Why not use `<style>` element?

- Causes bigger, confusing files
- Impossible to reuse between pages
- Annoying to edit

Later: "build" tools can output this

- But that's not the file we edit

Option 3: `<link>` a `.css` file (Do this)

In HTML:

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

In `example.css`:

```
#demo {
  color: red;
}

.selected {
  color: black;
  background-color: red;
}
```

Exceptions

Okay to use `<style>` element (in document)

- **If tools build it** for you
    - You don't suffer any of the downsides
    - Fewer requests

Okay to use `style` attribute (inline CSS)

- **If unknowable values assigned with JS**
- Values can't be defined by class names
    - Such as changing position by dragging
    - Values unknowable in advance

No hard rules on what to name `.css` file(s)

- Site size/organization
- May depend on libraries/frameworks used (if any)

Examples:

- `style.css` or `styles.css`
    - Often just a single common file
    - One page, or many pages all using same rules
- `home.css`/`index.css`, `about.css`, etc
    - Named for the page
- `header.css`, `menu.css`, etc
    - Rules for one part of page(s)

No hard rules on where to put the `.css` files

Common patterns:

- Same folder as any HTML files
- Subfolder where HTML is
- Document Root or Root subfolder

Put CSS in same folder as HTML

```
website/
-- some-private-file.txt
-- public/ (Document Root)
  -- index.html (Home Page)
  -- index.css
  -- contact.html (Contact Us Page)
  -- contact.css
  -- about/
    -- index.html (Main About Page)
    -- index.css
    -- staff.html (About Staff Page)
    -- staff.css
    -- location.html (About Location Page)
    -- location.css
```

Pro/Cons of putting CSS in same folder as HTML

- Meh: Both absolute and relative paths work well
- Pro: Easy to find CSS for given HTML file
- Pro: Can move HTML folders around easily
    - If you use relative paths
- Con: Messy if you have many files in folder
- Con: Harder to share CSS among many HTML files
- Con: Doesn't work well with dynamic pages
    - No "folder where HTML is"

Put CSS in subfolder where HTML is

`css/`, `styles/`, `media/`, `assets/`, etc subfolder

```
website/
-- some-private-file.txt
-- public/ (Document Root)
  -- index.html (Home Page)
  -- contact.html (Contact Us Page)
  -- assets/
    -- index.css
    -- contact.css
  -- about/
    -- index.html (Main About Page)
    -- staff.html (About Staff Page)
    -- location.html (About Location Page)
    -- assets/
      -- index.css
      -- staff.css
      -- location.css
```

Pros/Cons of CSS in subfolder where HTML is

- Meh: Both absolute and relative paths work well
- Pro: Can move HTML folders around easily
    - If you use relative paths
- Meh: Separates HTML files from CSS files
- Con: Harder to share css among many HTML files
    - But easy among HTML files in same folder
- Con: Doesn't work well with dynamic pages
    - No "folder where HTML is"

Put CSS in root or root subfolder

`/css/`, `/styles/`, `/media/`, `/assets/` root subfolder

```
website/
-- some-private-file.txt
-- public/ (Document Root)
  -- index.html (Home Page)
  -- contact.html (Contact Us Page)
  -- about/
    -- index.html (Main About Page)
    -- staff.html (About Staff Page)
    -- location.html (About Location Page)
  -- assets/
    -- about-index.css
    -- contact.css
    -- home-index.css
    -- location.css
    -- staff.css
```

Pros/Cons of putting CSS in root or root subfolder

- Meh: Absolute paths work better
- Pro: Can move HTML folders around easily
- Pro: Work fine with dynamic pages
- Meh: Separates HTML files from CSS files
- Con: Easier to share css among many HTML files
- Con: CSS file names must be clear
    - Easily more unrelated CSS files in that folder

Often sites will use multiple approaches

- Shared CSS placed in Doc Root subfolder
    - Loaded using absolute path by many pages
- CSS for 1 page placed in same/subfolder of HTML
    - Loaded using relative path by that page

```
website/
-- some-private-file.txt
-- public/ (Document Root)
  -- index.html (Home Page)
  -- index.css
  -- contact.html (Contact Us Page)
  -- contact.css
  -- about/
    -- index.html (Main About Page)
    -- index.css
  -- assets/
    -- styles.css
    -- menu.css
```

How many CSS files?

No set answer. Common to have:

- 1 file for site-wide standards
- 1 file for page-specific css

Sites might have 1 stylesheet, might have 5

- All about levels of abstraction and reuse

CSS in Practice

- Each Rendered **element** has **CSS properties**
- Each **property** = one rendered behavior aspect
    - Color, background-color, font-size, etc
- **Rules** are sets of **property** changes
- **Rules** apply to elements that match **Selectors**

CSS Rules

- A rule is **selector(s)** + **declaration(s)**
- Each **declaration** sets a **property**

```
p {
  color: rebeccapurple;
}

li {
  border: 1px solid black;
  padding: 0px;
}
```

More on declarations shortly

CSS Rule Errors are Silently Skipped

Invalid rules are skipped

- Next *identifiable* rule still runs
- No error message to user!

```
/* Notice missing `{` below! */
p
    color: rebeccapurple; /* broken rule */
}

p {
    color: lime;    /* all part of broken rule */
}

p {
    color: hotpink;  /* actually works */
}
```

Declaration Errors Silently Skipped

Invalid declarations are skipped

- Next *identifiable* declaration still runs
- No error message to user!

```
/* Notice missing `;` below! */
p {
    color: rebeccapurple   /* Missing ;, invalid */
    background-color: almond; /* Treated as part of above */
    font-size: 1.2rem; /* Works normally */
    padding: gibberish; /* invalid value, skipped */
    font-weight: bold; /* Works normally */
}
```

CSS Comments

- **Comments** in CSS appear between `/*` and `*/`
    - HTML comments are different!
        - `<!-- comment here -->`
- Comments ignored for rendering
- Used to communicate with humans
    - Ex: `/* Below fixes Safari bug with lists*/`
- Sometimes used to communicate with other tools
    - Ex: `/* eslint-disable-next-line */`

Avoid Poor Comments

Instructors, examples, and ChatGPT (et al):

- Will use comments to explain what a line does
- Great for people learning from the code

In "real" code such comments are bad

- Most code will already be clear
- Comments must be updated when code changes
    - Code changes often
    - Wrong comments worse than no comments

Do not have comments that repeat the code

Real Examples from Previous Assignments

```
a {
    color: white; /* font color */
}
```

```
header li {
    height: 100%; /* header and li same height */
}
```

```
#main {
    display: grid;
    grid-template-rows: 20% 80%;/* left side menu is 20% width */
}
```

```
/* Footer styles */
footer {
    color: black;
}
```

Using Comments Wisely

- **Only have when more benefit than hindrance**
- **Explain Why/Context**
  - Code says what, not why!
  - Ex: `/* Leaves space for floating nav */`
  - Ex: `/* overridding org defaults */`
- **Supply date or way to later lookup if still needed**
  - Ex: `/* Work around for bug 8675309 */`
  - Ex: `/* Edge breaks w/o (2023-12-13)*/`
- **Use for visible section labels, not code lines**
  - Ex: `/****** Navigation Styles *******/`
  - Even then, only when beneficial

Selectors

A rule has one or more comma separated **selectors**

- Declarations apply to any matching elements

https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Selectors

```
p, li {
  background-color: aqua;
}
```

Selector Matches

Which elements does a selector match?

- Tag name: `p {...}`
- "id" `#demo {...}`
- A class `.example {...}` (**most common**)
- Descendants `div .wrong {...}` (**notice space!**)
- Direct children `div > .wrong {...}`
- Attributes `[href] {...}`
- Many other options (read on MDN)

Match Elements By Tag Name

- `p {...}`
- `ul, ol {...}`

Used to set *default* appearance of ANY of that element

- Including future additions

Do not use to set particular appearance

Use Tag Name only for Default Appearances

Common Mistake to overuse Tag Name Selector

- When only have particular usage of element
    - So far!

Example:

- `ul` is often used to build navigation menus
- **You don't want menu to be default styling of `ul`**
- Even if navigation is currently your only use of `ul`

More Changes than is New

Remember: HTML is made up of nested elements

If element A has element B in element A's **content**...

- Element A is the **parent** of element B
- Element B is the **child** of element A

```
<p>a parent with <a href="/place">child</a></p>
```

- A **descendant** element is a child, grand-child, etc.
- Elements with the same parent are **siblings**
- An **ancestor** element has descendants

These terms and concepts will be used a lot

Match Elements by Ancestry

- `div p {...}`
- `li a {...}`

The final element is what is matched

- `li a {...}` styles the `a`, not the `li`
- Only `a` elements descended from `li` are matched
- Does not need to be a direct child
- Works with any combination of selector types
- Works with multiple levels (uncommon)
    - `ul li p a {...}`

Examples of Descendant Matching

`li a {...}`

- `<li>Unmatched <a href="/">Matched</a></li>`
- `<li><p>Unmatched <a href="/">Matched</a><li>`
- `<div><a href="/">Not matched</a></div>`

Longer sample:

```
<li>
  <a href="/">Matched</a>
  Not Matched
  <a href="/">Matched</a>
  <p>Unmatched<a href="/">Matched</a></p>
</li>
```

CSS Nesting allows for rules inside rules

- Treated as descendants
- Originally from SASS CSS Preprocessor
- Added to most modern browsers late 2023
- **This course won't use** (Not "Widely Available")

```
p {
    background-color: aqua;
    a {
        color: magenta;
    }
}
/* Same as above */
p {
    background-color: aqua;
}
p a {
    color: magenta;
}
```

Match Elements by `id` attribute

- `#demo {...}`

Put a `#` before the value of the `id` in selector

- The `#` is NOT part of the actual `id` value
- `<p id="demo">Matched</p>`

Not commonly used outside of examples by itself

- More when we get to `specificity`
- By definition matches at most one element
- Does get used to scope changes when combined...

Selectors can be combined

Use both `id` *and* `type`

- `p#intro {...}`
    - Type must be listed in selector first
    - No space between
    - Makes both required to match

Below selectors match different elements:

- `p#intro {...}`
- `p #intro {...}`

Upcoming selectors combine as well

Matching descendant of `id`

- `#my-section p {...}`
    - Pattern to set defaults within that ancestor
    - Upcoming selectors may also be descendant
        - Or ancestor

Different teams might "own" different page sections

- Scoping changes to your `id` prevents interference

Match elements by class name

- `.demo {...}`

Put a `.` before the value of the `class` in CSS

- The `.` is NOT part of the actual `class`
- `<p class="demo">Matched</p>`
- Matches if ANY of the classes match
- `<p class="demo other">Matched</p>`

Most commonly used!

- More when we get to `specificity`

Combining Class Selectors

Any number of combinations are possible

- `.demo.other {...}` - both classes
- `.demo .other {...}`
    - Descendant is a **combinator**!
- `.demo.other.example {...}` - many classes
- `p.demo {...}` - type and class
- `#root.demo {...}` - id and class

Match Direct Child Elements

Not any descendant, only direct child matched

- `div > .matched {...}`
- `div > p {...}`
- `#root > ul > li {...}`
- `div p.matched > a {...}`
    - `div p.matched` = descendant of div
    - `p.matched > a` = `a` that is child of that `p`

Example of Direct Child Matches

- `div.example p.matched > a {...}`

```
<p class="matched">
  Not Matched
  <a href="/">Not Matched</a>
</p>
<div class="example">
  <p class="matched">
    Not Matched
    <a href="/">Matched</a>
    <span>
        Not Matched
        <a href="/">Not Matched</a>
    </span>
  </p>
  <div>
    <p class="matched"><a href="/">Matched</a></p>
  </div>
</div>
```

Match Element by Attribute/Attribute Value

- More variations than seen here (see MDN)
- `[lang] {...}` - any element with `lang` attribute
- `img[alt] {...}` - any img element with `alt`
- `.active[lang] {...}`
- `input[type="text"] {...}` - text inputs
    - ONLY those explicitly set to "text"
    - Not those defaulting to "text"
- Can combine with any other selectors

Quick Note: Classes are most common selector

- We will discuss why later, but take note now
- Default to using class selectors
    - Unless you have a reason not to
    - This assignment limits your options
        - To force you to practice other selectors
    - But in future, prefer class names for selectors
        - Starting point, not hard rule

Repeat: Use class selectors as default starting point

What if nothing matches selector?

- No match = No error
- That rule isn't applied to any element

If element LATER matches

- Rule will apply to that element then
- Foreshadowing!

If element later no longer matches

- Rule will no longer apply to that element then

Casing is used to communicate

- Previously said **indentation** is used for humans
- So too is **casing**
    - When uppercase/lowercase letters
    - How multiple words are separated

A very common mistake

- New coders often treat as unimportant
- Your future team will reject your work
- Your future self will hate you

Different Casing Conventions (Part 1)

- `CONSTANT_CASE`
    - All uppercase
    - Words separated with `_`
    - Used to indicate "constants" in JS/Java/Python/etc
- `snake_case`
    - All lowercase
    - Words separated with `_`
    - Used in Python
    - NOT used in this course

Different Casing Conventions (Part 2)

- `MixedCase` / `PascalCase`
    - First letter of words capitalized
    - Words squished together/no separation
    - Used in some traditional coding languages
    - Used for components in Javascript (JS)
        - Also JS classes, distinct from CSS classes
- `squishedlowercase` (? No known name ?)
    - All lowercase
    - Words squished together/no separation
    - Used for *most* HTML attribute names

Different Casing Conventions (Part 3)

- `camelCase`
  - First letter of words capitalized, except first
  - Words squished together/no separation
  - Used in many traditional coding languages
  - Used in Javascript (JS)
- `kebab-case`
  - All lowercase
  - Words separated with `-`
  - Used for CSS property names
  - Traditionally used for HTML/CSS class names
  - Used for *certain* HTML attribute names

Casing systems we use in 6150

- `CONSTANT_CASE` - Javascript (JS) **constants**
- `camelCase` - Javascript (JS) **variables**
- `MixedCase` - Javascript (JS) **components**
- `kebab-case`
    - CSS/HTML **class names** (BEM allowed)
    - CSS **properties**
    - Certain HTML **attributes**
- `squishedlowercase` (I made this name up)
    - Most HTML **attributes**

CSS Declarations

"body" of a **CSS rule** is **declarations**

```
{
  some-css-property: value;
  another-property: value;
}
```

- Each declaration ends with semicolon (`;`)
- If **property** not known, declaration skipped
- If **value** is wrong, declaration skipped
- Indentation for Humans (but important!)
- Multiple lines for Humans (also important!)
- `p { color: lime; background-color: black }`
    - Valid!

Vendor Prefixes

Certain properties have **vendor prefixes**

- Example: `-webkit-transform-style: flat;`
- Older way of "trying out" new properties
    - Devs told not to use in production
    - Devs used in production
    - Now kept working forever
- Avoid these in modern CSS when possible
    - A few historical ones still exist

Shorthand properties

Some properties accept multiple values to apply to multiple properties:

```
p {
  border: 1px solid black;
}

p {
  border-width: 1px;
  border-style: solid;
  border-color: black;
}
```

Use these where the meaning is understood

Nothing wrong with being more explicit for clarity

Repeated Properties

If a property listed more than once in rule

- Latest *valid* value applies

```
p {   /* color will be red */
  color: white;
  color: red;
  color: does-not-exist;
}
```

Property Inheritance

Some properties are **inherited** by descendants

- Unless overridden
- Most colors and typography are inherited
- `p { color: rebeccapurple; }`

```
<p>Purple <span>Also Purple</span></p>
```

- Sizes and positioning are not inherited
- `ul { height: 250px; }`

```
<ul>
  <li>Not Each</li>
  <li>Size of Parent</li>
</ul>
```

CSS Custom Properties

Properties that start with `--` are user-decided

- Follow all rules of other properties
- Used similar to "variables"
- More later

What If?

If an element matches different selectors?

```
p {
  color: aqua;
}
.wrong {
  color: red;
}
```

Resolve via the **cascade**

CSS Cascade

Rules all browsers follow

- For each property of each element

Decide *which* CSS overrides other CSS

- **Origin** and **Importance**
- **Specificity**
- **Position**

A great summary: **https://2019.wattenberger.com/blog/css-cascade**

Cascade Origin and Importance

Origin

- Site css > user settings > browser defaults

Importance

- Does the declaration ends in `!important`
    - Confusing (does not mean "not important")
    - `p { color: red !important; }`
- `!important` declarations override non-important
- Origin weights reverse with `!important`
    - `!important` user css > `!important` site css
- **Don't use `!important`**

Why not to use `!important`?

- You add `!important` to override a declaration
- Then someone needs to override your override
    - They add `!important` to theirs
- **More Changes than is New**
- Soon: Half the CSS is `!important`
    - Seemingly randomly

Previous Best Practice:

- `!important` only to override library
- Otherwise use **specificity** (coming next!)
- Now have **cascade layers** to handle these

Cascade Specificity

Rules are of equal origin and importance?

- More **specific** selector wins
- "Jane" is more specific than "student"
- inline (`style` attributes) most specific
    - **Don't use `style` attributes**
- `id` > `class` > `type`

What color are One, Two, Three?

```
.example {
  color: magenta;
}

#example, #more {
  color: red;
}

p {
  color: lime;
}
```

```
<p id="example" class="example">One</p>
<p class="example">Two</p>
<p id="more">Three</p>
```

Specificity Example Answers

```
.example {
  color: magenta;
}

#example, #more {
  color: red;
}

p {
  color: lime;
}
```

```
<p id="example" class="example">One - Red</p>
<p class="example">Two - Magenta</p>
<p id="more">Three - Red</p>
```

Specificity Adds

Combined Selectors add totals to determine specificity

- But each (id, class, type) counted separately
- `.example.more` is more specific than `.example`
- `.example.more` is less specific than `#example`

Identical origins, importance, and specificity?

- Latest one wins!
- "Latest" = loaded later in the document

```
p {
    color: red;
}

p {
    color: dodgerblue; /* Wins over above */
}

.why-does.same-element {
    color: plum;
}

.have-so.many-classes {
    color: palevioletred; /* Wins over above */
}
```