CSS Overview

CSS provides

- Rules for appearance of HTML
- Based on structure

Stylesheets

There are a few ways to apply CSS to HTML

- inline CSS on element (don't do)
- `<style>` element (don't do)
- A stylesheet file linked via `<link>`

Inline CSS

(Generally don't do this)

CSS can be applied to an element as an attribute

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

Example

Why not use Inline CSS?

- Hard to override
- Impossible to reuse
- Really annoying to edit

Using a style element

(Generally don't do this)

```
<head>
  <style>
    #demo {
      color: red;
    }
  </style>
</head>
<body>
  <div id="demo">Example</div>
</body>
```

Example

Why not use style element?

- Makes for big files
- Impossible to reuse between files
- Annoying to edit

Using a stylesheet file

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

#demo {
  color: red;
}

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

How many stylesheets?

Varies, but typical to have:

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

Sites might have 1 stylesheet, might have 5

- all about level of abstraction and reuse

Exceptions

Okay to use style element

- if tools build it for you
- You don't suffer any of the downsides
- fewer requests

Okay to use inline CSS

- if assigned with JS *and*
- values aren't just class names
    - such as changing position by dragging

CSS Rules

CSS is made up of **rules**

- A rule is **selector(s)** and **declarations**

```
p {
  color: #C0FFEE;
}

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

invalid rules/declarations are skipped and the next rule/declaration tried.

Selectors

A rule has one or more comma separated **selectors**
https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Selectors

```
p, li {
  background-color: #BADA55;
}
```

- tag name: `p {...}`
- id `#demo {...}`
- a class `.example {...}`
- descendants `div .wrong {...}`
- direct children `div > .wrong {...}`
- many other options

Declarations

The "body" of a CSS rule is declarations.

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

If a property doesn't exist, the next will be tried

Browsers have specific properties with "prefixes"

- example: `--webkit-transform-style: flat;`
- you generally should avoid these in modern CSS

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

CSS colors

- A named color **https://drafts.csswg.org/css-color/#named-colors**
- a hexadecimal RGB color (e.g. `#BADA55`)
    - 3, 4, 6, and 8 character varieties
    - 3 or 4 have hex chars doubled
        - e.g. `#639` is `#663399`
    - 4 or 8 include alpha aka opacity
- `rgb()` or `rgba()` passing 3 RGB vals and an alpha
    - passed RGB values are decimal
    - alpha is 0-1 or 0%-100%
- non-RGB systems like `hsl()` or `hwb()`

What If?

If an element matches different selectors?

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

Resolve via **specificity**

CSS Specificity

- `!important` is the most specific (overrules all)
    - *Only* use this to override an external library
- Inline CSS is the next most specific
    - You should also not be doing this
- id selectors (`#example`) are next
- class selectors (`.example`) are next
- element selectors (`p`) are next

Selectors can combine to increase specificity

- `.example.wrong` is more specific than `.example`
    - still less specific than `#example`

Same Specificity?

If two selectors have the same specificity

- the winner will be the "most recent"
    - later in the file or page

Avoid Specificity War

If you have multiple sources of CSS

- the different sources will use specificity to override one another
- This can lead to "specificity wars":
    - one source will make a selector more specific
    - but that breaks another place
    - so the source of the other place raises THEIR specificity
- There is only pain and tears in a specificity war
    - avoid by having a way to target each "level" of source

Scoping on a shared page

A semi-common pattern:

- Your content container has an id
- Use classes (not ids) for lower levels
- Use `#YOUR-ID .YOUR-CLASS` as your CSS pattern

Means you only have to have one unique id per source of content

- Ensures your class styling won't impact outside your content