

# CSS Overview

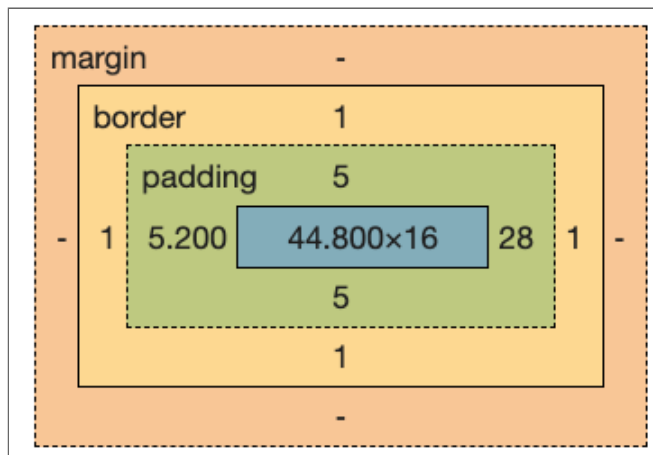
CSS provides

- Rules for appearance of HTML
- Based on structure

# CSS Box Model

Every rendered element is a "**box**":

- content has **height** and **width**
- **padding** around it has size in four directions
- a **border** on all four sides has a width
- **margins** between the border and adjacent boxes has a width in four directions



# Box Sizing

How wide is the below element?

```
p {  
  width: 100px;  
  padding: 10px;  
}
```

- With `box-sizing: content-box;` (default) = 120px
- With `box-sizing: border-box;` = 100px;

Common to see:

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

# 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: #COFFEE;  
}  
  
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](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()`



# Property Inheritance

Some properties on parents are inherited by children unless overridden

- Some other properties are not inherited
- Ex: "color" is inherited
- Ex: "width" is not inherited
- Most colors and typography are inherited
- Sizes and positioning are not

# What If?

If an element matches different selectors?

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

Resolve via **specificity**

# CSS Specificity

- `!important` is the most specific (overrides 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

# Emmet

- Editor may have "snippets"
  - define expansions of known content
- Emmet is a generic standard
  - for HTML and CSS (and lorem ipsum text)

**<https://docs.emmet.io/>**

# Lorem Ipsum

## Fake text

- taken randomly from an old latin speech
- see how a layout looks with "text-like" content
- Real content is always better
  - But rarely available at design time
- Many tools to generate "lorem text"
  - in-editor or websites to cut/paste



# CSS Units

- `%` of container
- `vh` and `vw`
  - "viewport"
- `px` vs `rem` vs `em`
  - `px` is (mostly) fixed
    - fixed is often bad
  - `em` causes inheritance problem
  - sizes based off of "root" font "em" width
    - "root" is `<html>` element
  - <https://css-tricks.com/html-vs-body-in-css/>
  - `rem` useful with browser text settings

# CSS Variables

Often we have values that we want to reuse

- height/widths of elements interacted with (nav?)
- colors (background, accent, highlight, etc)

Technically these are "custom properties", and follow the normal cascading/precedence rules

# Outside CSS

CSS took a long time to add variables

We will talk about SASS and LESS later, they have their own solutions

- But SASS/LESS aren't actual CSS

This is the pure (but limited) CSS solution

# Using a CSS Custom Property

Assign:

```
.some-selector {  
  --my-var: black;  
  --another: 5rem;  
}
```

Use:

```
p {  
  color: var(--my-var);  
}
```

"Global" assign:

```
:root {  
  --main-bg-color: #BADA55;  
}
```

# Pseudo-classes

Added to a selector to indicate a state

- `:hover`
- `:focus` and `:focus-within`
- `:active`
- `:not()`
- `:first-child`
- `:nth-child()`

# Pseudo-elements

Not elements, but allow you to style them like one

- `::selection`
- `::first-line` and `::first-letter`
- `::before` and `::after`
  - These require a `content` property

# CSS Properties

- `filter`
  - `filter: brightness()`
- `opacity`
- `font-family`
- `visibility`
  - hides without removing from layout
  - can be good/bad for accessibility

# CSS Functions

- `calc()`
- `max()` and `min()`
- `clamp()`
  - 3 args, preferred should be a value that changes



# Media Queries

- Wraps CSS Rules
- Rules applied or not based on query
- Says if the rules are matched

# Screen Width

If CONDITION, apply CSS rules

```
@media (min-width: 1000px) {  
  body {  
    background-color: red;  
  }  
}
```

# Reduced Motion

- options are `no-preference` or `reduce`
- which involves less work?

```
@media (prefers-reduced-motion: no-preference) {  
  .my-element {  
    animation: flashy-zoom-in-out 1s;  
  }  
}
```

# Orientation

- If you care past width...

```
@media (orientation: portrait) {  
  body {  
    display: flex;  
    flex-direction: column;  
  }  
}
```

# Printing

A deep rabbithole

- Alternative to generating PDFs
- Not always the best alternative

```
@media print {  
  h3 {  
    page-break-before: always;  
  }  
}
```

# Summary - CSS Purpose

CSS provides rules for appearance

- based on structure
- where structure matches rules, appearance applies

# Summary - Box Model

Every element is a "box" of "boxes"

- content width and height
- padding width and height
- border width and height
- margin width and height

`box-sizing` property

- `content-box`: `width` and `height` are content
- `border-box`: `w + h` are content+padding+border?

# Summary - Stylesheets

CSS added to your page:

- inline in elements
  - rare except for specific needs
  - can't reuse
- in `<style>` element
  - rare without tools
  - can't reuse
- as a separate `.css` file
  - via `<link>` element with `href` attribute
  - common
  - multiple CSS files when different reuse cases



## Summary - Rules

- Rules are selector(s) + declarations
- invalid rules skipped over

# Summary - Selectors

- comma separated
- if any selectors match, declarations applied
- symbols indicate type of selector, no symbol = element selector
- Connected symbols = must match all:
  - `div#root.active`
    - "div with id root and class of active"
- space = descendant, easiest to read backwards
  - `div .wrong`
    - "element with a class of wrong that is a descendant of a div"

# Summary - Declarations

- kebab-case, each ends in semicolon
- "prefixes" (`--webkit-*`) mostly retired
- "shorthand" properties set multiple properties
  - use where understandable
  - avoid where confusing (be explicit then)
- Color values can be
  - RGB 3,4,6,8 hex characters starting with `#`
  - `rgb()` or `rgba()` (decimal values)
  - `hsl()` or `hwb()`
  - transparency/opacity is "alpha"
    - 0.0-1.0 or 0%-100%

# Summary - Cascade

All matching rules are applied

- some properties are inherited from parent element
  - generally text and color related properties
  - not size, display, or layout related properties

# Summary - Specificity

If a property gets two different values, which takes effect?

- both do, but one is overridden
- Selector Specificity
  - which rules have properties overridden
  - `!important` > inline > id > classes > element
- Same specificity:
  - most "recent" overrides
  - order of file loading
  - place in file