Department of Computer Science
University of California, Los Angeles

Computer Science 144: Web Applications

Spring 2025
Prof. Ryan Rosario

Lecture 4: April 9, 2025

Outline

1. CSS Review
2. CSS Layout
3. Responsive Web Design
4. Working with CSS in Production
5. UI Frameworks

1 CSS Review

2 CSS Layout

3 Responsive Web Design

4 Working with CSS in Production

5 UI Frameworks

CSS Review

Today we will discuss **layout** and responsive design. Last time we discussed **style** which was an earier step in browser parsing.

This topic is usually covered later in the term, but I feel we should get CSS out of the way.

I expect there will be a wide variety of experiences with today's content.

CSS Review (contd.)

Disclaimer: This lecture is dense. I do not like to discuss too much syntax and am trying to present *concepts*, but we will need to discuss syntax.

There resources within the slide deck that allow you to dig into the syntax. There is no substitute for hands-on practice.

CSS Review (contd.)

CSS is a set of rules and properties for specifying document formatting, styling and presentation.

1. Remember, `Rule = Selector + Declaration block`
2. The declaration block is a series of `"property:  value"`

A Few CSS Best Practices

CSS styles target elements, classes, IDs and other things (pseudoclasses and pseudoelements):

1. "Like styles" should be included as classes and IDs.
2. Targeting tags is discouraged.
3. Use semantic tags from HTML5 instead of `div`.
4. These semantic tags do not have default styles, but some validators enforce rules.
5. `span` should contain a class, ID, or inline style. Each `should` be different so targeting `span` is not appropriate.
6. Outside of HTML5 semantic tags, use `<div>`s liberally to play it safe.
7. Often a class or ID selector is enough.

A Few CSS Best Practices (contd.)

Some more:

1. Keep your selectors simple.
   - Often a `class` is enough
   - Shorter selectors = fewer surprises in cascade
2. **Prepare** for the future.
   - It's not a good idea to target a tag that is used once.
   - What if you use it again? Do you really want the same style?
3. Count on inheritance and the cascade
4. Wrap similar elements in a container if possible

CSS Custom Properties

You can create custom properties as "constants" and then use them in your styles. They are inherited. Declare them at the highest level where they will be used, such as <body>.

```
:root {
    --light-background-color: white;
    --dark-background-color: purple;
    --sidebar-width: 250px;
}  /* all descendants of body inherit these */

code { background-color: var(--light-background-color);  }
p {
    background-color: var(--dark-background-color, black);
    width: calc(100% - var(--sidebar-width));
}
```

Other CSS Things: Color

To be complete, there are a few ways of specifying color in CSS.

1. Color names
2. `rgb(red, green, blue)` where each parameter is 0 to 255
3. `rgba(red, green, blue, alpha)` where alpha is transparency 0 to 1, 1=opaque
4. Hexadecimal color (e.g. #rrggbb) (calculator)

Click here for UCLA colors .

1 CSS Review

2 CSS Layout

3 Responsive Web Design

4 Working with CSS in Production

5 UI Frameworks

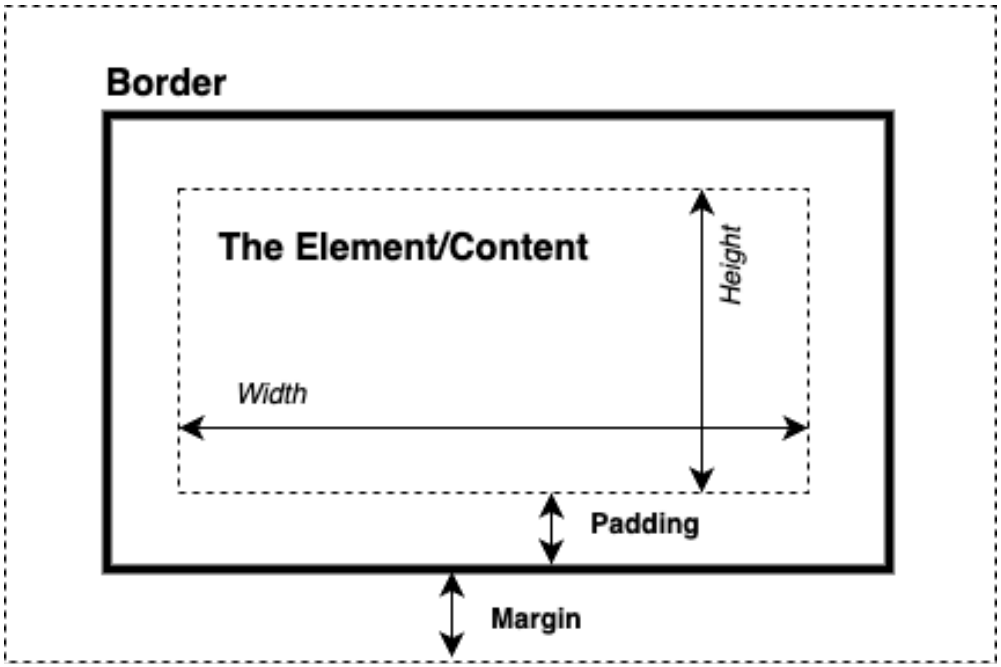
CSS Layout

CSS is not just about style — it's about presentation, which includes layout.

This requires a few features of CSS:

1. the CSS box model
2. block vs. inline elements
3. the `display` and `position` property

CSS Box Model



Each HTML element (represented by the innermost dashed lines, is surrounded by a series of boxes all with different purposes. Aside from the border, none of these are styleable.

CSS Box Model (contd.)

Borders, margins and paddings can involve the entire box or only certain sides: top, right, bottom and left.

e.g. `margin-top`, `padding-left`, `border-bottom`.

Borders are special. They can have width, color, and style and we can use a shorthand:

CSS Box Model (contd.)

```
p { border: 5px solid red;  /* width style color */ }
```

is syntactic sugar for this mess:

```
p {   border-left-width: 5px;
      border-right-width: 5px;
      border-top-width: 5px;
      border-bottom-width: 5px;
      border-left-style: solid;
      border-right-style: solid;
      border-top-style: solid;
      border-bottom-style: solid;
      border-left-color: red;
      border-right-color: red;
      border-top-color: red;
      border-bottom-color: red; }
```

CSS Box Model (contd.)

For margin, and padding you can also do this (different example):

```
p { margin: 1px; }
/* 1px around the element */

p { margin: 1px 2px; }
/* 1px on the top and bottom, 2px on the left and right */

p { margin: 1px 2px 3px; }
/* top right bottom */

p { margin: 1px 2px 3px 4px; }
/* top right bottom left */
```

This **does not** work for `border`.

CSS Box Model is Quirky

The box model is quirky though:

1. **Inline elements:** top/bottom margins and padding do not work. Use `line-height` to set vertical spacing.
2. **Margin collapsing:** If there are multiple elements vertically with different margins, the maximum value is the margin that is used.
3. **Default margins:** `p` (1em), `h1` (0.67em), `body` (8px), `div` (0)
4. **Negative margins:** margins can be negative to overlap with a previous element or to counteract another element's margin.
5. **Auto margin:** use `auto` to horizontally center an element, cannot vertically center.

I've run into every one of these problems and features.

Overflowing Containers

We have discussed space around elements. Sometimes elements, namely containers, are so stuffed with text that they overflow.
What do we do?

Overflowing Containers (contd.)

Suppose we specify that a container has a specific height and width and the text doesn't fit in it. We can specify what happens when it `overflows`.

- `visible`: text spills outside of the element. This is ugly.
- `hidden`: overflowed text is clipped — not shown to the user. This can also be ugly and user unfriendly, but there are times when it can be cleaner when resizing the screen
- `scroll`: always show a scrollbar, even if the text fits
- `auto`: show scrollbar only if overflow, don't otherwise
- `clip`: relatively new. Same as `hidden` but programmatic scrolling is disabled.

We can the behavior specific to horizontal and vertical: `overflow-x`, `overflow-y`.

Overflowing Containers (contd.)

We can also specify how we wrap long words within a text container with `overflow-wrap`:

- **normal**: try to break words on spaces, otherwise overflow the container.
- **anywhere**: long words will break if they overflow the container, but the break may not make sense and looks ugly.
- **break-word**: breaks a long word onto a new line, but this isn't perfect. A really long word will look more like "anywhere" behavior. This is usually what we expect in a book.

Layout Units

Example	Units
2px	Pixels
1mm	Millimeters
3cm	Centimeters
0.2in	Inches
12pt	Point ($\frac{1}{72}$ inch)
1em / 2em	Units relative to the computed or root font size
%	Percentage of the parent/container element
25vh	Percentage of the viewport height
50vw	Percentage of the viewport width

1px = 0.75pt assuming a 96dps display.

Page Flow

Remember that **block** level elements start and end with a newline, can have a height/width, and take up the entire container. *These elements are arranged from top to bottom on the page.*

Inline elements appear next to each other, cannot have a height/width, and only take up the space they need. *These elements are arranged from left to right.*

These behaviors dictate the **flow** of a page as it is rendered.

Overriding Page Flow using `display`

We can *override* how an element behaves in page flow using the `display` property.

- **block**: element has a new line before and after it, it a height/width that can be set, and takes up the entire space horizontal space of the container.
- **inline**: element appears on the same line as the previous element. No top/bottom margin, but maybe left/right. No ability to set width/height and only uses space it needs.
- **none**: hide the element entirely. It does not take up space. This is different than `visibility: hidden;` as hidden maintains space for the hidden element.
- **inline-block**: similar to `inline` (same line) but can have height/width and top/bottom/left/right margins.
- **grid** and **flex**: these are new display types that allow for more complex layouts. We will cover them in a bit.

positioning an Element w/r/t Page Flow

We can also specify the element's position relative to the normal document flow.

1. `static` (default): The element is rendered in the normal flow with no changes to position.
2. `relative`: we can use the `top, left, right, bottom` properties to shift an element's position relative to its position in the normal flow, while still remaining in normal flow.
3. `absolute`: The element is removed from the normal document flow (no space is reserved for it) and is positioned relative to its closest positioned ancestor (i.e., an ancestor with `position` other than `static`). If no such ancestor exists, it is positioned relative to the root HTML element. Sibling elements are ignored in layout.
4. `fixed`: position to a fixed part of the screen.
5. `sticky`: remains in flow and scrolls normally until it reaches a threshold, then sticks to that position until scrolled back.

Overlapping Elements

But when we move elements all over the screen, it's possible to end up with overlapping elements.

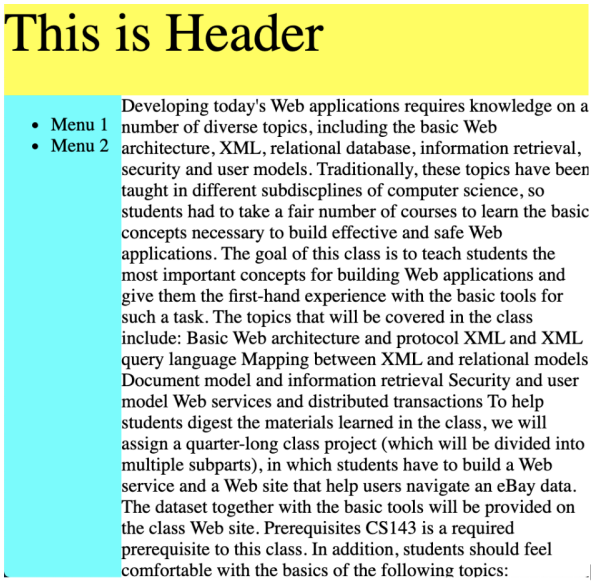
The `z-index` property specifies the vertical location if elements overlap (especially when it is intentional).

Higher `z-index` elements are placed on the top of lower elements.

Exercise

Checkpoint: Let's construct this simple layout together.

1. Header stays at the top
   - `width: 100%`
   - `height: 90px`
2. Menu stays on the left
   - width: 100px
   - height fills in all area beneath
3. Content area fills rest
   - Show scrollbar if overflow



Vanilla CSS is Limiting

So with vanilla CSS, we can style font and text styles, control element spacing, and create simple layouts.

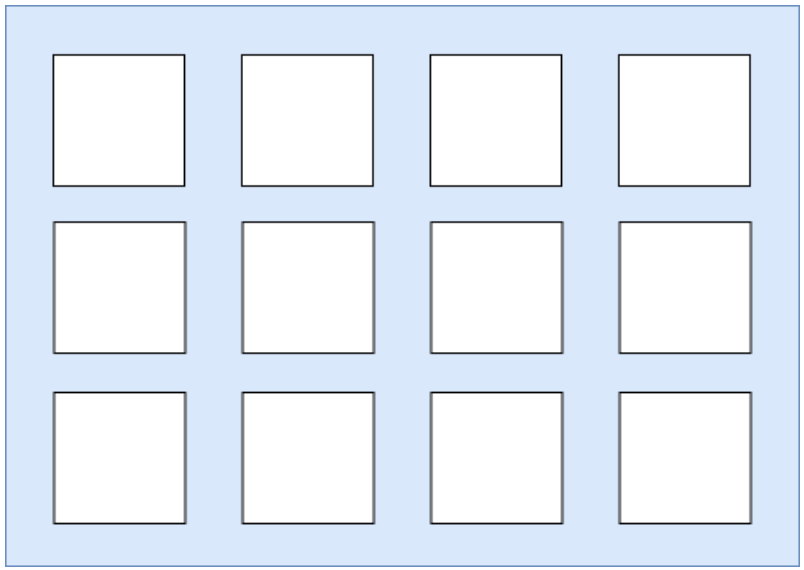
But some things are hard or impossible:

- Vertically centering elements
- Navbars with left and right actions
- Grid layouts
- Static footer at bottom of a window

`display: float` was originally used for layout but is now considered outdated for that purpose. Nowadays there are some additional display properties we can use...

CSS Grid

Pages, or containers on pages, can be laid out using a grid of rows and columns in a tabular arrangement. The full syntax is pretty tedious. We will cover the basics.



The dimensions are specified with `grid-template-rows` and `grid-template-columns`.

CSS Grid (contd.)

```
#grid {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-rows: repeat(3, 2em);
  gap: 0.5rem;
}
```

```
<div id="grid">
  <div></div>
  <div>Math</div>
  <div>ELA</div>
  <div>Student 1</div>
  <div>720</div>
  <div>560</div>
  <div>Student 2</div>
  <div>340</div>
  <div>490</div>
</div>
```

	Math	ELA
Student 1	720	560
Student 2	340	490

CSS Grid (contd.)

When cells have to be of different sizes, or span in complex ways, CSS Grid gets much more complicated.

You should read this guide.

CSS Flexbox

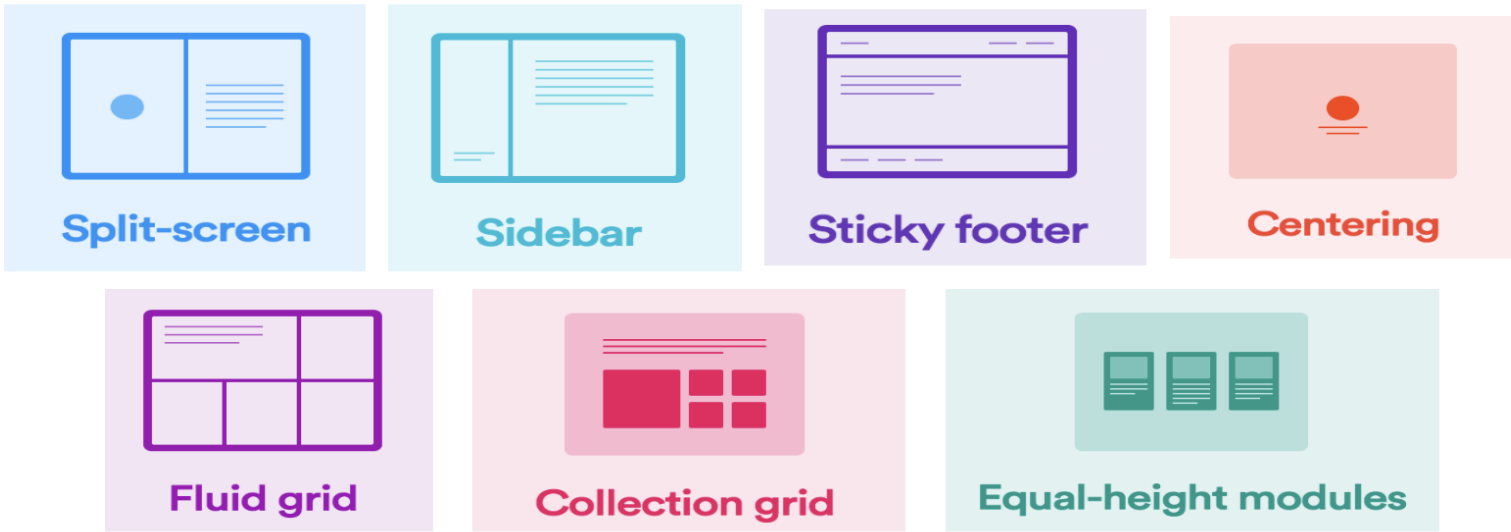
Flexbox is a "newer" addition to CSS. While CSS Grid allows a 2-dimensional layout, Flexbox can only display content as a series of rows, or a series of columns, not both in the same container.

But, different parts of your page can use different variants of Flexbox, and others can even use Grid if you wish.

Elements are dynamically resized or rearranged based on available space.

CSS Flexbox (contd.)

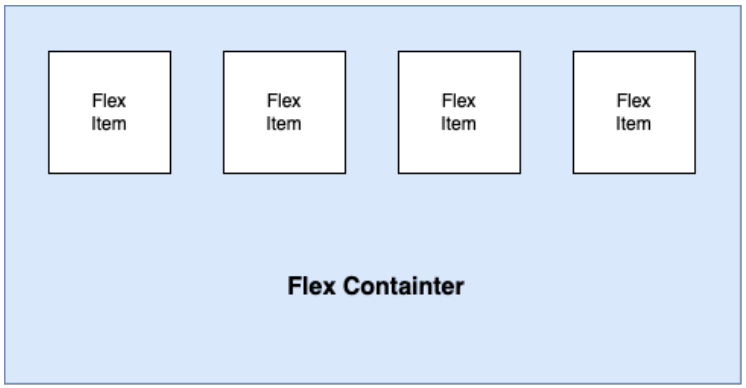
Flexbox allows a lot of beautiful designs that we see everyday.



Source: Stanford University

CSS Flexbox (contd.)

Flexbox changes how elements are laid out. The parent itself becomes a **flex container**. The direct children become **flex items**.



`flex-direction: row;`

`flex-direction: column;`

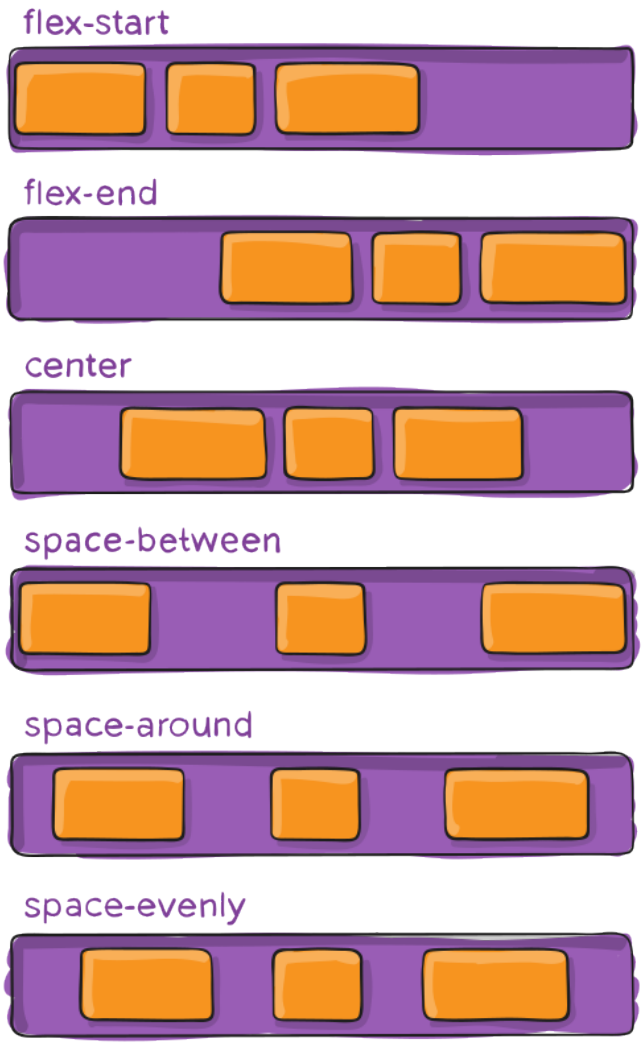
CSS Flexbox (contd.)

By default, flex items change their size to fit available space. That is, they are **responsive**.

For more information, click here.

For an interactive tutorial that dives more into syntax, click here.

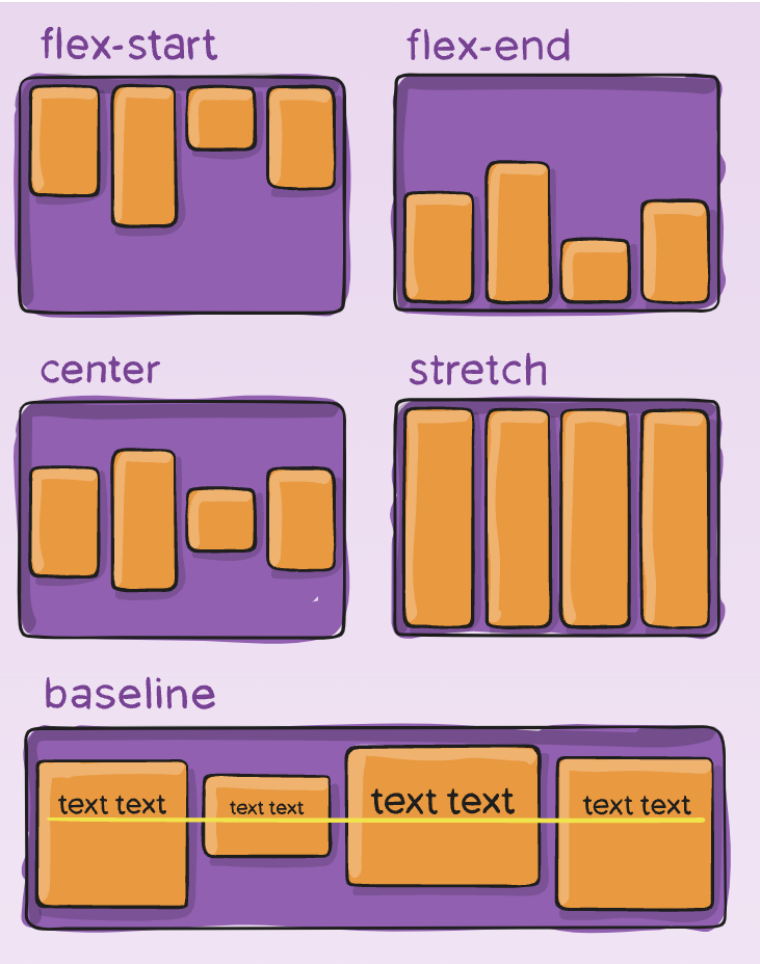
Useful Flexbox Properties



Flex items are justified within the space along the main axis (`flex-direction`).

How they are justified is controlled by `justify-content`.

Useful Flexbox Properties (contd.)



`align-items` controls the layout along the cross axis (opposite of `flex-direction`).

CSS Flexbox

You will want to read this guide to Flexbox.

1 CSS Review

2 CSS Layout

3 Responsive Web Design

4 Working with CSS in Production

5 UI Frameworks

Back Then

Back in the day, websites were build for this thing:

Now

Now we have these, and they all have different screen sizes:

Laptop

Smartphone

Tablet

The Point

The Point: We do not want to build N versions of each web app, one for each device.

Responsive Web Design allows us to develop pages that look nice on different screen sizes and devices.

The Point (contd.)

"Content is like water. The web app should flow into and fill whatever device you have."
-– Mendel Rosenblum, Stanford

"Water and electricity do not mix."
— Ryan Rosario, UCLA

The Point (contd.)

Responsive web design is a combination of:

- Fluid grids
- Flexible images
- Media queries

How should a site behave on a phone vs a laptop?

Fixed vs. Fluid

In a fixed layout

- Elements have fixed width
- Resizing the window does not change their sizes or arrangements

Fluid layout

- Elements use "percentage" of the width of the page

Responsive Web Design

Some general rules for responsive web design:

1. Do not force users to scroll horizontally. **Think about why.**
2. Do not force fixed-width elements. **Think about why.**
3. Do not force users to zoom in and out to read text. **Think about why.**

Responsive Web Design: Viewport

The first step towards a responsive design is the viewport.

The **viewport** is a portion of the web site that is visible to the user on their device screen. It should be readable. The web page may be large, but the user only sees the viewport.

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

Responsive Web Design: Viewport (contd.)

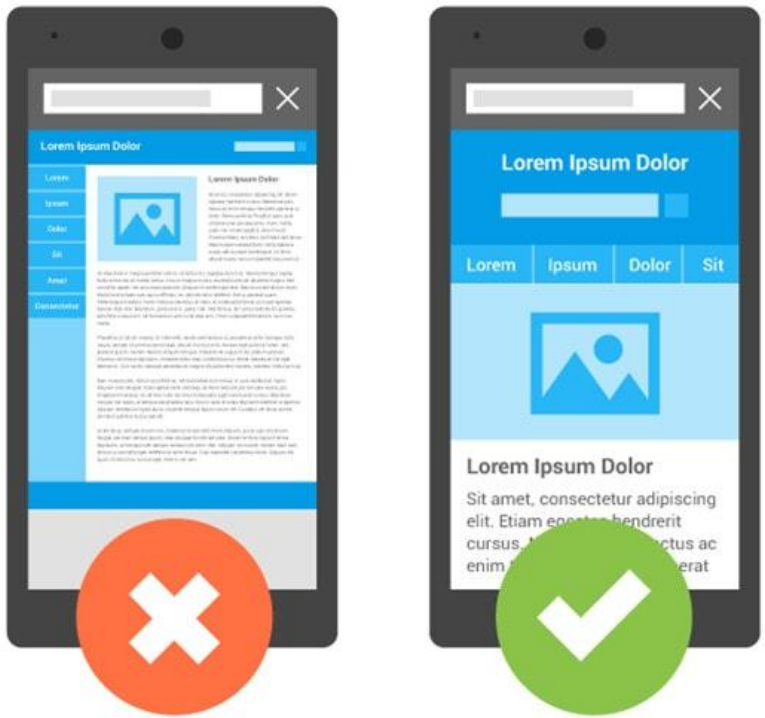
Why do we have to do this? By default, a mobile browser likes to pretend that it has access to a large screen ($>$ 980px). It doesn't.

`width=device-width` reminds the browser that it is a small screen and that the viewport (what the user sees) should be equal to the screen size.

`initial-scale=1` sets the initial zoom level to 1. Not zoomed in or out.

The page may now need to be scrolled. But there is a better way.

Responsive Web Design: Viewport (contd.)



Responsive Web Design: CSS Media Queries

This does not do much. We still need to apply custom CSS rules for specific devices.

```
@media (max-width: 800px) {
        /* CSS rules */
}
```

Enclosed rules are apply only if condition is true (`max-width: 800px` here). The condition can be a complex boolean condition.

Responsive Web Design: CSS Media Queries (contd.)

In the module *CSS Media Queries v4*, there are three media types:

1. `screen`
2. `print`
3. `speech`

In v3 there were many more and they have been deprecated.

Responsive Web Design: CSS Media Queries (contd.)

We can also check against a series of media features:

- orientation (portrait and landscape orientation on phones)
- `min-width`, `max-width` of media
- `min-height`, `max-height` of media
- `resolution`
- `device-aspect-ratio`
- `color` (bitrate)
- `scan` (specific to TV monitors)

Can connect into a Boolean expression with `and`, `or`, `not`.

Responsive Web Design: CSS Media Queries (contd.)

Photo gallery demo

Department website demo

Responsive Web Design: CSS Media Queries (contd.)

For more information on media queries and responsive web design, see here.

Responsive Web Design: CSS Media Queries (contd.)

Images and media should use percentages (%) or `vw` rather than fixed units so that they can grow and shrink with the viewport.

You should develop using a mobile first approach. If it looks good on mobile, it will look good on desktop.

1 CSS Review

2 CSS Layout

3 Responsive Web Design

4 Working with CSS in Production

5 UI Frameworks

Beyond CSS

Manually writing CSS like we have been doing is fine. In production, we now use other tools to simplify this process.

There are too many to mention, but two that stand out are **Sass** and **Tailwind**.

Sass Preprocessor

Syntactically **A**wesome **S**tyle **S**heets:

1. is an extension to CSS
2. is CSS pre-processor
3. lets you use features that do not exist in CSS, like variables, nested rules, mixins, imports, inheritance*, built in functions, and other stuff.

Let's walk through a couple of features...

Sass Preprocessor (contd.)

Sass is a CSS **preprocessor**.

This means we write our styles in some syntax (in an `.scss` file) and then *transpile* it into vanilla CSS.

This extra syntax adds a lot of flexibility to create reusable styling components.

Sass Preprocessor (contd.)

We can create **variables** for styles:

```
/* define variables for the primary colors */
$primary_1: #a2b9bc;
$primary_2: #b2ad7f;
$primary_3: #878f99;

/* use the variables */
.main-header { background-color: $primary_1; }

.menu-left { background-color: $primary_2; }

.menu-right { background-color: $primary_3; }
```

Sass Preprocessor (contd.)

We can have **nested rules**. These two are equivalent.

```
/* Sass */
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }
  li {
    display: inline-block;
  }
  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}
```

```
/* Vanilla CSS */
nav ul {
    margin: 0;
    padding: 0;
    list-style: none;
}
nav li {
    display: inline-block;
}
nav a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
}
```

Sass Preprocessor (contd.)

Properties can also be nested:

```
font: {
  family: Helvetica, sans-serif;
  size: 18px;
  weight: bold;
}

text: {
  align: center;
  transform: lowercase;
  overflow: hidden;
}
```

Sass Preprocessor (contd.)

Mixins allow us to group properties together that can be imported into other declarations:

```
/* Sass */
@mixin important-text {
  color: red;
  font-size: 25px;
  font-weight: bold;
  border: 1px solid blue;
}

.danger {
  @include important-text;
  background-color: green;
}
```

```
/* yields */
.danger {
  color: red;
  font-size: 25px;
  font-weight: bold;
  border: 1px solid blue;
  background-color: green;
}
```

Sass Preprocessor (contd.)

Variables can also be passed to mixins:

```
@mixin bordered($color, $width) {
  border: $width solid $color;
}

.myArticle {
  @include bordered(blue, 1px);  // Call mixin with two values
}

.myNotes {
  @include bordered(red, 2px); // Call mixin with two values
}
```

Tailwind

Tailwind takes a different approach. Tailwind is a utility-first CSS framework. It's opinionated and uses single purpose utility classes. The developer specifies these single purpose classes in their HTML and the CSS is generated from it.

Basically, the developer includes class names for each element using a specific grammar and Tailwind automatically knows how to style the element(s) according to them.

Tailwind (contd.)

The following HTML...

```
<!DOCTYPE html>
<html lang="en">
<head>
...
<link
href="https://cdn.jsdelivr.net/npm/tailwindcss@^3.0/dist/tailwind.min.css" rel=
</head>
<body>
  <div>
    <button class="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4
      Click me!
    </button>
  </div>
</body>
</html>
```

1 CSS Review

2 CSS Layout

3 Responsive Web Design

4 Working with CSS in Production

5 UI Frameworks

Popular UI Frameworks

There are many UI frameworks for CSS with various features and use cases. Some of the more popular ones include:

1. Bootstrap: Can be used with Sass. Provides a lot of styles to make generating UI components and experiences easier.
2. Materialize based on Google Material UI.
3. MaterialUI based on Google Material UI, but for React only.