CSS

- CSS provides the ***style*** and ***layout*** of a web page.

- There are a few ways to include CSS into an HTML page:
    - The preferred way is to include an external stylesheet (i.e. a CSS file) into a page using the `<link>` element within the `<head>` element:

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

    - You can also include a `<style>` element as a child of `<head>`.
        - This can be useful when you are unable to include a separate CSS file, e.g. when you are working within a CMS like WordPress.
        - Not as useful as an external stylesheet for a web site because the CSS needs to be repeated for each HTML file being styled.
    - You can also use inline styles by including a `style` attribute in the element you want to style:

```
<h1 style="color: blue">My Header</h1>
```

        - Don't do this unless you absolutely have to.
            - It's terrible from a maintenance standpoint, since you could have to change the same thing multiple times per document.
            - It mixes your presentation (CSS) with your structure and content, which makes your code difficult to read and understand.

CSS syntax

- CSS is comprised of a set of rules, each of which contains:
    - A set of **properties** with values that indicate how the HTML content is displayed.
    - A **selector** that indicates which HTML to apply the CSS properties to.

- For example, here's a CSS rule that would make all the <h1> elements on our page have orange text on a black background with an orange border:

```
h1 {
     color: orange;
     background-color: black;
     border: 1px solid orange;
}
```

    - Here, `h1` is the selector and the contents of the following block are the properties (before the colon) and their values (after the colon). Each property ends with a semicolon.

- To apply CSS rules to an HTML page, the browser matches the selectors against the DOM tree.

- Just like with any other code, use whitespace and newlines to make your CSS readable.
    - Use comments, too, when they are helpful:

```
/* This is a CSS comment */
```

Element selectors

- The simplest kind of selector uses element types, like we saw above:

```
p {
     Color: green;
}
```

- You can combine multiple elements in the same selector to be more specific. Each successive element in the selector applies to elements that are *descendants* of the previous element in the selector:

```
p em {
     color: red;
}
```

    - The above selector would apply to any `<em>` element that is a descendant of a `<p>` element, i.e. it would apply to the `<em>` tag in both of the following HTML snippets:

```
<p>Here's some <em>emphasized</em> text.</p>
```

```
<p>
     Here's some <strong>strongly
     <em>emphasized</em></strong> text.
</p>
```

Class selectors

- You can also apply a `class` attribute to any HTML element and then use the given class(es) in CSS selectors.
    - Individual classes are separated by spaces.

- The below `<p>` element is given two classes, `user-address` and `primary`:

```
<p class="user-address primary">...</p>
```

- We can use class names in CSS selectors by using a `.` in front of the class name:

```
.user-address {
     padding-left: 30px;
}

.primary {
     font-weight: bold;
```

```
}
```

- Just like in element-based selectors, you can have multiple classes, or a combination of classes and elements in a selector.
    - Each successive item in the selector designates a descendent of the previous one.

ID selectors

- You can also apply an `id` attribute to any HTML element and then use it in CSS selectors.
    - The `id` value is used in a selector by prepending a `#`.

- Here's an example of an `id` being used to style an element:

```
<div id="main-content">
     ...
</div>

#main-content {
     margin: 20px;
}
```

Combinators

- We've seen one way to combine sub-selectors to more specifically match HTML elements, there are actually a few different ways:
    - `AB` – any element matching both `A` and `B` at the same time, e.g.:
        - `p.user-address`
    - `A B` – any element matching `B` that is a descendent of an element matching `A`, e.g.:
        - `p em`
    - `A > B` – any element matching `B` that is a direct child of an element matching `A`, e.g.:
        - `ul > li`
    - `A + B` – any element matching `B` that is the next sibling of an element matching `A`, e.g.:
        - `.nav-header + .nav-item`

    - `A ~ B` – any element matching `B` that is among any of the next siblings of an element matching `A`, e.g.:
        - `.nav-header ~ .nav-item`

- You can also apply the same block of properties to multiple selectors easily by separating the selectors with a comma:

```
div.title,
div.subtitle {
      ...
}
```

Specificity

- A given HTML element might match many CSS rule selectors, so several rules might each set a given property on that element.

- ***Specificity*** of selectors is the main way the browser determines which property value takes precedence and should be appiled.

- Specificity is a measure of how specific a selector is, in terms of how many elements it could match.

- In general, element selectors are the least specific, class selectors are the next most specific, and ID selectors are the most specific.

- A 4-digit specificity weight is calculated for each selector. You can think of the digits of this weight as thousands, hundreds, tens, and ones (just like a 4-digit number).

- The factors of the specificity weight are calculated as follows:
    - Thousands: 1 if the matching selector is defined in a `<style>` element or in a `style` attribute. Otherwise 0.
    - Hundreds: add 1 for each ID in the selector.
    - Tens: add 1 for each class, attribute, or pseudo-class in the selector.
    - Ones: add 1 for each element type or pseudo-element in the selector.

- The properties of the CSS rule whose selector has the highest specificity take precedence for a given HTML element.

- In the result of a tie in specificity, CSS rules defined *later* in the source code take precedence.

Cyclomatic complexity and CSS selectors

http://csswizardry.com/2015/04/cyclomatic-complexity-logic-in-css/

- In software engineering, ***cyclomatic complexity*** is a metric that describes the complexity of a program or function.
  - It measures the number of independent paths through the code.

- As engineers, we strive to *reduce* cyclomatic complexity because high complexity means:
  - Code is harder to reason about.
  - Code is harder to maintain, modify, or reuse.
  - There are more potential failure points.
  - Code can have unintentional side effects.
  - Code is harder to test.

- We can think of each item in a CSS selector as increasing the cyclomatic complexity of that selector.
  - In particular, each item in the selector results in an *if* statement the browser needs to run.
  - For example consider this selector:

```
main section div.social-buttons
     .facebook-button a span {
     ...
}
```

  - This would result in the following logic executed in the browser:

```
if (is span)
     if (inside a)
          if (inside .facebook-button)
               if (inside .social-buttons)
                    if(on div)
                         if (inside section)
                              if (inside main) {
```

```
                                        // do something
                                    }
```

    - We would never actually write code like that (hopefully).

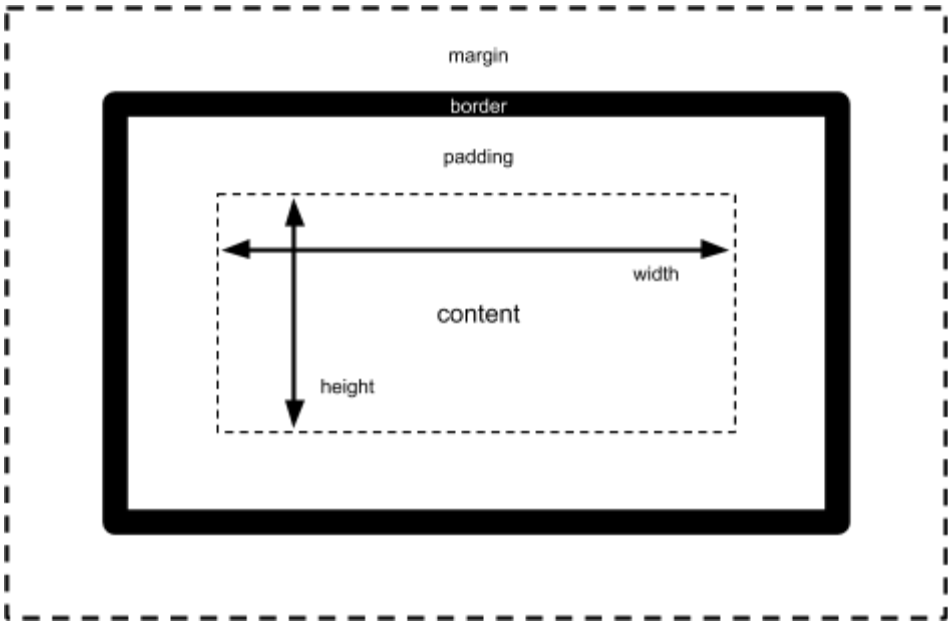
    - Instead, we'd write something like this:

```
if (is .fb-btn-link-text) {
     // do something
}
```

- You should think about the logic that results from your CSS selectors and strive to keep the cyclomatic complexity low.
    - Keep only the logic that's really needed in your selectors.
    - Create new classes to help you reduce the logic.
    - Think about your selectors from the right side first.
    - Make sure your selector logic is *intentional*, not just that it works.
        - E.g. use `.nav-list` instead of `header ul`.

The box model

- Every element in an HTML document is structured as a many-layered box:



- We can adjust the layers of this box with the following corresponding CSS properties:
    - `width` and `height` – the width and height of the content box, which includes the box's content as well as the boxes of it's child elements:

      ```
      width: 200px;
      height: 300px;
      ```

        - Can also set `min-width`, `min-height`, `max-width`, and `max-height`.

    - `padding` – the layer between the outer edge of the content box and the border:

      ```
      padding: 20px;  /* all sides same padding */
      padding: 0 10px;  /* no padding top and bottom;
                           10px left and right */
      padding: 5px 10px 20px 8px;  /* 5px top;
                                      10px right;
                                      20px bottom;
                                      8px left */
      ```

    - `border` – the potentially visible layer between the outer edge of the padding and the inner edge of the margin. By default it's invisible, but it can be given a thickness, style, and color:

      ```
      border: 1px solid black;
      ```

    - `margin` – the outer area surrounding the box, which sits against the other boxes in the layout:

      ```
      margin: 20px;  /* all sides same margin */
      margin: 0 10px;  /* no margin top and bottom;
                          10px left and right */
      margin: 5px 10px 20px 8px;  /* 5px top;
                                     10px right;
                                     20px bottom;
                                     8px left */
      ```

- Some important properties of boxes:
    - The background (set, e.g. by background-color) extends to the outer edge of the border.
        - This can be changed with the `background-clip` property.
    - Content that becomes larger than the box will overflow and cause scroll bars to appear.
        - You can control this behavior with the overflow `property`.
    - The `height` property cannot be assigned a percentage value.
        - Box heights always adopt the height of the content, unless an absolute height value is set.
    - Borders can't have percentage width settings, either.
    - By default, `width` and `height` control the size of the content box.
        - You can adjust this using the `box-sizing` property.
    - Many of these components have properties that allow you to control the individual sides separately, e.g.
        - `padding-left`, `padding-right`, `padding-top`, and `padding-bottom`.
        - `margin-left`, `margin-right`, `margin-top`, and `margin-bottom`.
        - `border-left`, `border-right`, `border-top`, and `border-bottom`.

CSS box types

- We looked at block and inline elements when studying HTML.

- The box type can actually be controlled using the `display` CSS property:
    - `display: block` – the box is stacked upon other boxes, with newlines separating it from the others.
    - `display: inline` – the box flows with the surrounding text, with no newlines separating it from other inline elements.
        - Breaks lines just like regular text.
        - `width` and `height` properties do not apply.
        - `padding` property affects background and border placement, like with `block` boxes, but only affects surrounding elements/text within the inline flow (i.e. on the left and right of the box).
        - `margin` property only affects surrounding elements/text within the inline flow (i.e. on the left and right of the box).
    - `display: inline-block` – a hybrid display

        - Makes the box flow with the surrounding text without inserting newlines, like an inline box.
        - Can be sized using `width` and `height` and maintains its block integrity.
        - `padding` and `margin` behave like with block boxes, even within the inline flow.

Styling text

- There are lots of different CSS properties that affect text styling.

- `font-family` is used to control the font itself.
    - You can specify a single `font-family` property:

      ```
      font-family: Helvetica;
      ```

    - You can also specify a stack, where the browser falls back on the elements in sequence a particular element couldn't be found:

      ```
      Font-family: "Helvetica Neue", Helvetica,
           sans-serif;
      ```

    - Web-safe fonts available in all browsers.

    - There are also generic fonts:
        - `sans-serif`
        - `serif`
        - `monospace`
        - `cursive`
        - `fantasy`

- `color` controls the color of the font
- `font-size` controls the size of the font
- `font-weight` controls the weight of the font, i.e. the boldness/lightness
    - e.g. `font-weight: bold`
    - e.g. `font-weight: 600`
- `font-style` controls styling aspects of the font
    - e.g. `font-style: italic`

- `text-transform` can apply a transform to all the text
  - e.g. `text-transform: uppercase`
- `text-decoration` can apply a decoration to the text
  - e.g. `text-decoration: underline`
  - e.g. `text-decoration: none`
    - Useful for removing underline from links.
- `text-shadow` can apply a shadow to the text
- `text-align` can justify or align text left, right, or center
- `line-height` controls the height of a single text line
  - e.g. `line-height: 1.5`

Values and units

- The various CSS properties each takes value arguments of different kinds.

Numeric values

- Many different CSS properties take numeric arguments. There are a few types of numeric arguments:
  - Length and size
    - Used for layouts, typography, and more.
    - Lots of different units:
      - Absolute units (always the same size):
        - `px` (pixels)
          - Probably the most popular length unit.
        - `mm`, `cm`, `in`
        - `pt` (points: 1/72 inch)
      - Relative units:
        - `em`
          - Width of a capital M in the current font-size.
        - `vw`, `vh`
          - 1% of viewport width and height, respectively. Not always supported.
      - Percentages can also be used for many CSS property values.
  - Colors:
    - Keyword colors, e.g. `red`, `orange`, `yellow`, `blue`, `green`, `purple`, `black`, `white`, `gray`.
    - Hexadecimal values: `#123abc`.

            - Each pair of values represents one of the RGB channels.
        - Functions:
            - `rgb(255, 0, 255)`
            - `rgba(255, 0, 255, 0.5)`
                - Allows transparency and blending.

Positioning

- We can control the position of an HTML element using the CSS `position` property.
    - `position` lets us control the *type* of positioning.
    - The `top`, `right`, `bottom`, and `left` properties let us specify the actual position.

- We refer to an element as ***positioned*** if its `position` property value is `relative`, `absolute`, or `fixed`.

- There are several different types of positioning, corresponding to different values of the `position` property.
    - `static` – the element behaves normally and is laid out in its current position in the flow.
        - This is the default value of the `position` property.
    - `relative` – all elements are laid out as if the element was not positioned, and then the element's position is adjusted, relative to its original position.
        - This leaves a gap where the element was originally positioned.
    - `absolute` – the element is removed from the normal positioning flow (thus leaving no gap) and then is positioned relative to its closest positioned ancestor (or to `body`).
    - `fixed` – the element is removed from the normal positioning flow (thus leaving no gap) and then is positioned relative to the whole viewport.
        - The element does not move when scrolled.

Pseudo-elements and pseudo-classes

- There are two types of special CSS selectors that can be used to style elements only in a certain state or to style only certain parts of elements.
    - A ***pseudo-class*** is a particular keyword preceded by a colon (`:`) that you can append onto another selector (just like a regular class) to specify that

it should be selected only when the element is in a certain state, e.g. when it is being hovered over.
    - A ***pseudo-element*** is a particular keyword preceded by two colons (`::`) that you can append onto another selector to style a certain part of that element.

- There are several important pseudo-classes you should be aware of:
    - `:hover` – this is used to select an element that is being hovered over with the cursor. For example, this will style a link that is being hovered over:

```
a:hover {
    ...
}
```

        - You can also use this to select a descendant of an element that is being hovered over (you can use most pseudo-classes in this way), e.g. this will style a link within an element with class `button` that's being hovered over:

```
.button:hover a {
    ...
}
```

    - `:focus` – this is used to select an element that has focus (e.g. from the user selecting it with a keyboard or mouse). For example, this would style a link with focus:

```
a:focus {
    ...
}
```

    - `:first-child` and `:last-child` – these allow you to select elements that are either the first or last child of their parent element in the DOM. For example, this would style both the first item and the last item in an unordered list:

```
ul li:first-child,
ul li:last-child {
    ...
```

```
}
```

    - There are many more than are listed here. You can use Google to find them.

- There are also several important pseudo-elements you should be aware of:
    - `::after` – this allows you to add a virtual last child of the selected element, e.g. to add some cosmetic content. For example, you could use it to add an arrow after a link:

```
a::after {
     content: "↗";
}
```

    - `::before` – this is like `::after` and allows you to add a virtual first child of the selected element, e.g. to add some cosmetic content.

    - `::first-letter` – this allows you to apply CSS text styling properties to the first letter in the first text line of a block.

    - `::first-line` – this allows you to apply CSS text styling properties to the first text line of a block.

Advanced box styling

- There are several additional properties for styling CSS boxes that don't fit neatly into any of the categories above, but which you should be aware of.

- The `box-shadow` property allows you to add a drop shadow to a CSS box. Its essential form takes 4 argument values:
    - The X offset of the shadow.
    - The Y offset of the shadow.
    - The blur radius of the shadow.
    - The color of the shadow.
    - These look like this:

```
box-shadow: 5px 5px 10px rgba(0, 0, 0, 0.25);
```

- The `border-radius` property allows you to round the corners of a box:

  ```
  border-radius: 10px;
  ```

  - This works even if no border is specified for the box by rounding the corners of the background.

- The `float` property allows you to take an element out of the normal layout flow and to place it along the left or right of its container.
  - Text and inline elements will wrap around floated elements.
  - Implies `display: block` for the floated element.

- The `cursor` property allows you to change what the mouse cursor looks like when it's hovering over an element. For example, to make the cursor look like a pointing hand over an element, you could do this:

  ```
  cursor: pointer;
  ```

Advanced layout with flexboxes

- Flexboxes make lots of layout tasks much easier, including some that were previously either difficult or impossible to do in a convenient, flexible way, e.g.:
  - Vertically centering a block within its parent.
  - Making all children take up an equal (or proportional) amount of the available height or weight.

- In a flexbox layout, we choose which elements we want to lay out as flexible boxes and then set the `display` property of their parent item to have value `flex`:

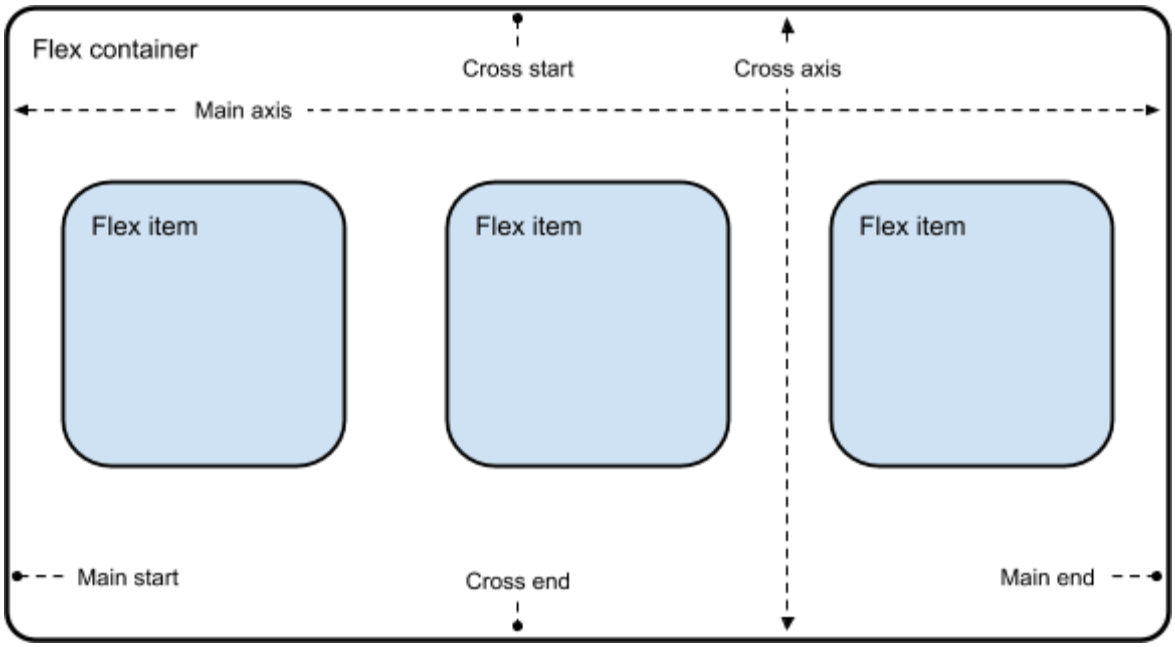
  ```
  display: flex;
  ```

  - This parent container is called the ***flex container***.
  - The flexible box elements are called ***flex items***.

- Flexible box elements are laid out along two main axes:
  - The ***main axis*** is the axis running in the direction in which the flexible box elements are being laid out (i.e. in rows across or columns down the page).

        - The start and end of the main axis are called the ***main start*** and ***main end***.
    - The ***cross axis*** runs perpendicular to the direction in which the flexible box elements are being laid out.
        - The start and end of the cross axis are called the ***cross start*** and ***cross end***.

- This all looks something like this:



- We can specify the direction of the main axis (i.e. whether the flex items are laid out in columns or rows) using the `flex-direction` property:

```
flex-direction: column;
flex-direction: row;  /* This is the default */
```

- If you run out of width or height in your parent container, the children will overflow and break the layout. You can instead make them wrap using the `flex-wrap` property:

```
flex-wrap: wrap;
```

- You can control how much space the flex items take up using the `flex` property on the flex items themselves.

    - You can assign two (or one, or three) values to `flex`, one specifying the amount of space the element will take up and one specifying the minimum width
    - For example, if this was assigned to all of the flex items, they would all take up an equal amount of space and have a minimum width of 200px:

      ```
      flex: 1 200px;
      ```

        - The first value is known as the `flex-grow` factor.
        - The second is the `flex-basis`.

    - You could also make one of your elements take up more space by assigning a larger `flex-grow` factor than you assign to the other flex items.
        - The amount of space given to a flex item is the ratio of its `flex-grow` factor to the sum of all `flex-grow` factors.

- You can align flex items along both the main axis and the cross axis using these properties:
    - `align-items` – controls where the flex items sit on the cross axis.
        - Some possible values:
            - `center` – center items on the cross axis
            - `flex-start` – align items at the start of the cross axis
            - `flex-end` – align items at the end of the cross axis
    - `justify-content` – controls where the flex items sit on the main axis.
        - In addition to the values mentioned above for `align-items`, you can also use these values:
            - `space-between` – distribute all elements evenly along the main axis
            - `space-around` – like `space-between`, but also leaves space on either end of the main axis

- You can nest flexboxes to any level for extremely flexible layouts.

- Check caniuse.com to see if you can use flexboxes in your browser (hint: you probably can!).

Responsive layout

- Modern web sites are viewed on all kinds of different devices, and it's important for a site to look good on all of those devices.

- A CSS layout is called ***responsive*** if it adapts well to all the different kinds of devices it is viewed on.

- A full exploration of responsive layouts is beyond the scope of this course, but one essential thing to be aware of when implementing responsive layouts is the CSS `@media` rule.

- The `@media` rule allows you to apply different blocks of CSS rules when the device/viewport on which the user is viewing your site meets certain conditions.

- One simple way to use `@media` is to apply different CSS rules at different viewport widths.

- For example, the below set of `@media` rules would color the background of the header differently as the viewport was resized from 400 pixels to above 1200 pixels:

```
@media(max-width: 400px) {
     header {
          background-color: green;
     }
}
@media(max-width: 800px) {
     header {
          background-color: red;
     }
}
@media(max-width: 1200px) {
     header {
          background-color: blue;
     }
}
```

- CSS can be applied in lots of different ways using a simple `@media(min-width)` setup like this. Imagine the possibilities!