HTML and CSS

Topics

- HTML
- CSS Basics
- Position
- Grid
- Rem and Em
- Grid Areas

Intro to HTML

- HTML is simply text that the browser knows how to read (parse).
- There is the *metadata* that is located in the `<head>` tags.
- And there is the *visible* part of the HTML, located in the `<body>`
- The broswer represents the HTML as a *tree structure*.

```
<body>
    <header>
        <div class="icon">
            <p>icon</p>
        </div>
        <nav>
            <ul>
                <li> about </li>
                <li> projects </li>
                <li> contact </li>
                <li> social media </li>
            </ul>
        </nav>
    </header>
    <main>
        <div class="element"></div>
        <div class="element"></div>
        <div class="element"></div>
    </main>
    <footer>
        <div class="wrapper">
            <div> &copy </div>
            <div> 2019 </div>
            <div> made by me </div>
        </div>
    </footer>
</body>
```

- Here the `body` is the top level 'node' of the tree.
- That parent node has three children, `header`, `main`, and `footer`.

- Those three (`header`, `main`, `footer`) are on the same 'level' and are *'siblings'* to each other
- The `header` element (or node) has two children: a `div` with the class of `icon`, and the `nav` element.
- The `nav` element has only one child, as does the `div`.
- The `ul` element has four `li` children. Each of these `li`s are siblings to each other.
- The above HTML could be (partially) represented in a tree structure like this:

```
body
  |
  |- header
  |    |- div (.icon)
  |    |   |- p
  |    |
  |    |- nav
  |        |- ul
  |           |- li
  |           |- li
  |           |- li
  |           |- li
  |- main
  |    |- div
  .    .
  .    .
  .    .
```

- *Any* type of tag can be used to be a child or parent of any other - although some combinations make more sense than others.
- Indentation is *extrememly* important.
- Also worth noting is that `index.html` is the document that a server will deliver by default. For this reason we tend to set up a new folder with a new `index.html` inside it. If we *'Go Live'* with several folders around, including several index.html files, then odd things can happen. To keep things clean it's best to open a new VS Code instance containing only the project directory that you are working from. It's more work up front, but this is traded off against more work later.

CSS

- We talked about setting up our project, and here we are adding CSS to the project in `styles.css`.
- Our folder structure now looks like this:

```
/my-project
      |- index.html
      |- styles.css
```

- We add our CSS in by including it into the metadata in the `<head>` section of the HTML:

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

- We started by straight away writing our CSS into this file, although we could write it in the HTML (known as *in-line CSS*)
- There are two ways that we discussed of targetting an HTML element directly, with an id, or with a class.
- An *id* can only be used for one element in the HTML.
- To target it in the CSS we use a # prefix. Here is an example:

```
#first-main {
    background-color: navy;
}
```

- A *class* can be used on multiple elements within the HTML.
- We target a class with the . prefix. Here is an example:

```
.first-list {
    background-color: darkcyan;
}
```

- Probably the central couple of features for understanding CSS are:

1. That everything on the page by default wants to move up into the top left corner of the screen.
2. That there are three main types of elements: block, in-line, and in-line block.
    - block: takes up the whole width of the page.
    - in-line: takes up only the width of the content (and cannot set width and height, amongst other things)
    - in-line block: is like in-line in terms of content, but can be adjusted like block.

This StackO article explains this well

- (I glossed over this second point in class when I talked of only two types of element. Both of the points above are simplified and summarised to form the basis of understanding CSS from a top level. It's more complex, but we can see the kinks as they arise.)
- We looked at a couple of basic CSS attributes:
    - *background-color*: the colour of of the background to the element
    - *margin*: how far things external to the element should stay away from the border of the element
    - *padding*: how far internal content should stay away from the border of the element
- For the last two we used the analogy of the room, with margin acting like spikes on the outside of the room, pushing things away, and padding being like spikes on the inside, keeping things inside the room away from the walls.
- Keep in mind that css is based around content, and that elements will often have no height until you change the height attribute in the CSS.
- More and more you need to think in percentages and ratios. As much as you can manage, keep clear of fixed pixel quantities for elements. Of course you can use them, but percentages and ratios

should be the first stop where posssible.
- CSS is frustrating. It takes a long time to become very good, but you can become capable in a short period of time if you take care with the basics.

Flexbox

- Flexbox is a modern CSS tool.
- Flexbox helps in creating structure within your page (more simply than in the past)
- Do *not* simply use Flexbox for everything because it is a new technology.
- Flexbox is most effective for:
    1. Aligning items in one direction
    2. Centring an item
- There are many possible uses for Flexbox, but you should ask yourself why you need it for the particular task at hand.
- The biggest issue for understanding Flexbox is that *the Flexbox commands are placed as attributes on the parent to the children that are being moved by the flex commands*.
- This means that if you have a `ul` and several `li`s, and you are hoping to arrange the `li`s in some way using Flexbox, then the commands to do that are attributes on the `ul`.
- Putting Flexbox on to an element makes these children `in-line` (technically `inline-block`, but I'm trying to keep things simple for now).
- Remember that because of the above point, you will need to give these items enough space to work within. By default they will only take up the space of the content, so if you want to create space, the 'wrapper' needs to be expanded to give them space to work within.
- Flexbox is very helpful when it comes to responsive design (websites that respond well to changes in the view size on different devices and browser sizing). It isn't a panacea, but a very good starting point.
- There are many Flexbox commands, but the most useful are these:
    - `display: flex;`
        - This sets the element as the Flexbox wrapper, and applies the Flexbox commands to the children.
    - `flex-direction: row`
        - You can set the direction as `row` (across the screen) or `column` (up and down the screen).
    - `justify-content: space-around`
        - This sets the position and spacing of the flexed items in the same direction as the flex-direction.
        - More explicitly, if the flex direction is `row`, this will control the items left and right (across the page). If it is `column` it will control their position and spacing up and down the page.
        - There are many commands here. `flex start` pushes them up one end, and `flex-end` down to the other. You can space them evenly or push them right apart.
    - `align-items: center`
        - This command is the compliment to the previous, but aligns the content along the perpendicular axis. That is, it will push the content around in the opposite direction from the one in which they are lined up.
- There are many more commands, and different levels of complexity. My tendency is to see how far I can get with Flexbox first, and then make other interventions later.

Position

- We talked about the default setting for CSS elements being `static`. In a way this means that no position has been set.
- The default will be for the elements to move up the page into the top left corner, but keeping in mind the `block`, `inline` and `inline-block` aspects of each element.
- Now we are looking at ways to move the elements around away from this default or standard position (`static`).
- As with all of these things (and perhaps moreso in this instance) these are concepts that are hard to comprehend in theory, and that you need to tinker with to fully understand.
- I will put some code at the bottom that you can use to investigate.
- As always, please come to me with any questions.

Relative

- The first we looked at is *relative* position. The idea here is that you are moving the element, but relative to where it would have been in the normal *flow* of the elements.
- So if you put `position: relative;` on an element, and `top: 50px;`, you are moving it 50 pixels down away from where it would be positioned ordinarily.
- `left: 50px;` would move it right 50 pixels to the right, away from the left edge where the element would be in `static` mode.
- `top`, `bottom`, `left`, and `right` are used to position the element.
- Once again, *relative* means relative to where the element would have been.

Absolute

- *Absolute* position is like relative, but operates relative (yes, sorry) to the nearest *positioned* parent. If there is no parent in that element's parental tree structure, it will be positioned relative to the body, the ultimate parent to any visible element.
- By default, any element within a parent will want to move to the top left corner.
- If that parent has a position set on it (anything but the default, `static`), then this element can now be moved relative to its parent by adding the attribute `position: absolute`.
- The `top`, `bottom`, `left`, and `right` offsets are all in play again here.

Fixed

- The easiest of the three that we discussed today. *Fixed* position places that element relative to the body (using the same offsets), and does not move the element regardless of scrolling or any other non-fixed element.
- The element is taken completely out of the normal flow of CSS elements.
- The typical use for this is in navigation menus.

Example code

- In the example code you can see a couple of instances of two classes being placed on one element, eg:

```
<div class="inner relative"></div>
```

- This is very useful when you need several elements to have the same class, but then distiguish some of these elements.
- We also saw an example of more fine-tuned targetting of elements using CSS (although not in the code below). For example:

```
.electronics-wrap h1 {
    font-size: 90px;
}
```

- The above code will target only the `h1`s that are children of the `electronics-wrap` element.
- No other `h1` elements will be affected.

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <link rel="stylesheet" href="./styles.css">
    <title>Position CSS</title>
</head>
<body>
    <div class="first">
        <div class="inner"></div>
        <div class="inner relative"></div>
        <div class="inner"></div>
    </div>
    <div class="second">
    </div>
    <div class="fourth">
        <div class="third">
            <div class="fifth">
            </div>
        </div>
    </div>
</body>
</html>
```

CSS

```
* {
    margin: 0;
    padding: 0;
```

```
}

body {
    /* background-color: aquamarine; */
}

.first {
    display: flex;
    flex-direction: row;
    justify-content: space-around;
    align-items: center;
    background-color: brown;
    height: 200px;
    width: 100%;
}

.inner {
    background-color: aquamarine;
    height: 150px;
    width: 150px;
}

.second {
    position: fixed;
    bottom: 0px;
    right: 0px;
    background-color: blue;
    height: 100vh;
    width: 30px;
}

.third {
    /* position: static; */
    background-color: blueviolet;
    height: 150px;
    width: 150px;
    position: absolute;
    top: 30px;
    left: 30px;
    z-index: 3;
}

.fourth {
    position: relative;
    background-color: black;
    height: 1000px;
    width: 300px;
}

.fifth {
    height: 80px;
    width: 80px;
    background-color: blanchedalmond;
    position: absolute;
```

```
    bottom: 20px;
    right: 20px;
    z-index: 2;
}

.relative {
    position: relative;
    top: 50px;
    left: 50px;
    z-index: 1;
}
```

Grid

- We discussed CSS Grid and how it can be used to line things up in two directions, both across the page, and up and down the page.
- Grid is a powerful tool, but should not be used just because it is shiny, new, and exists. It is am implement to get a particular job done, so it's important to think about why you have decided to use Grid.
- In the same way as with Flexbox, you put the Grid commands *on the parent* to the items that are being placved on the grid.
- There is much to discuss with Grid, but to keep it simple, first we want to have a way to set up a static grid.

```
.grid-container {
    display: grid;
    grid-template-columns: 50px 1fr 3fr 2fr 100px;
    grid-template-rows: 200px 200px 200px 200px 200px;
    grid-auto-rows: auto;
    grid-gap: 4px;
}
.item {
    padding: 15px;
    background-color: green;
    border-radius: 5px;
}
```

- Here we are setting up a 5 by 5 grid.
- In this instance, the rows are the simplest to discuss. The `grid-template-rows` attribute is set so that each of rows will be 200px.
- The columns here are more complex. The first column is `50px` wide, and the fifth column is `100px` wide. These values are fixed.
- For the middle three columns we are using fractions. They will take up the remaining space across the available width, and each will take up the proportion that they have been given.
- There are a total of 6 fractions available (1 + 3 + 2 = 6). The first of these columns will take up 1/6 of that space. The second will take up 3/6, or 1/2 of the space. And the third will take up 2/6, or 1/3.

- We can also use fractions on rows. The difference is that with the web, the width is generally the more fixed value, whereas the height tends to be more dependent on the amount of content (although this is not always true).
- There is also the `repeat` command. For example:

```
.grid-container {
    /* ..other grid code.. */
    grid-template-columns: repeat(5, 200px);
}
```

- Here we will get five 200px columns, which is the same as `grid-template-columns: 200px 200px 200px 200px 200px;`
- We can also repeat fractions:

```
.grid-container {
    /* ..other grid code.. */
    grid-template-columns: repeat(3, 1fr 2fr);
}
```

- In this case we will get three times that 1/2 ratio.
- Repeat is a function that takes two arguments: the first is for the number of times to repeat, and the second is the pattern to be repeated.

Heights

- In the first example above we are setting a very static 5 by 5 grid that will not change. If there are fewer or greater than 25 items then we may have specified too few or too many grid spaces for them. We might prefer to have a way to cope with as many items as are thrown our way.
- Instead of being prescriptive about how many rows we expect up front, we could simply declare the height of any rows created:

```
.grid-container {
    /* ..other grid code.. */
    grid-template-rows: 100px;
}
```

- This will set the value of any row that is created at 100px.
- We might want the heights to to `auto` so that they take on the size of the content:

```
.grid-container {
    /* ..other grid code.. */
    grid-template-rows: auto;
}
```

- We have also discussed how there is much discussion in Grid of *explicit* vs *implicit*.
- We have not gone into depth on these concepts. In the above examples we are always being explicit - that is, we are explicitly specifying how we want our grid to look.
- This article gives a good overview of the difference between these concepts, and how that relates to building your grid. If you are interested, please have a look: Grid: explicit v implicit

Gaps

- The last basic command is `grid-gap` and this is simply the gap that you would like between items.

```
.grid-container {
    /* ..other grid code.. */
    grid-gap: 5px;
}
```

- This command puts a gap of 5px between items.

Minmax

- `minmax` is a command that can be simple to understand conceptually, but tougher to get your head around in specific cases. Tinkering with this command to try to understand it is a good idea.
- Here is a classic example to examine:

```
.grid-container {
    /* ..other grid code.. */
    grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));
}
```

- What we are saying here is that we want to repeat a pattern. We want that pattern to auto-fit to the screen (only repeat as many times as needed), and that each element take up a minimum of 250px across, and 1fr.
- The two arguments to the minmax function are the minimum and maximum values of the items.
- The more natural way to understand this is that we are saying for the browser to create as many columns as will fit, but that each should have a minimum size of 250px. When there is room for more columns, create more columns (and each should take up the same amount of space, `1fr` each of the total fractions available).
- This means that the number of columns will change depending of the width of the window.
- There are many variations that can be used for this command, but start with the command above, and tinker with it. As you need different and more complex commands you can consult the docs.

Summary

- Please use grid for the job as required.
- Grid is very powerful, even with the basic simple commands above.

CSS

Rem and Em

Rem

- This is a way to steer clear of fixed sizes (eg, `px`), and have your webpage sizes based on ratios.
- `rem`s are the more global version. Your base value is the size of the font in your `html` (the *root* of the document). This is 16px by default.
- This means that you can set all sizes relative to this size, and then if you decide to resize things on your website, you can change this *root* size, and all of the other sizes will change as well.
- This means that they will hold their relative sizes, and that they don't go out of sync (the border sizes will change with the font size and the margin size, etc). *Em*
- `em`s are a similar concept, but they are ratios based around the size of the font in the *current* element.
- This *could* be the same size font as the root, but need not be.
- This means that you can design a particular element, and have it exist as a self-contained set of values - values that make internal sense.
- This element could be moved around and retain it's internal consistency.
- Here is some code to have a play with:

```
* {
    box-sizing: border-box;
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
}

html {
    font-size: 50px;
}

body {
    margin: 200px;
    font-size: 20px;
}

h1 {
    /* font-size: 80px; */
    font-size: 3rem;
}

h2 {
    /* font-size: 30px; */
    font-size: 2rem;
}

h3 {
    /* font-size: 16px; */
    font-size: 1rem;
}

button {
    border: 0.1em solid;
    padding: 1em;
```

```
    font-size: 1.5rem;
}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <link rel="stylesheet" href="./styles.css">
    <title>Rem and Em</title>
</head>
<body>
    <h1>
        This is My Big Heading
    </h1>
    <h2>
        Subheading of Sorts
    </h2>
    <h3>
        In publishing and graphic design, Lorem ipsum is a placeholder
text commonly used to demonstrate the visual form of a document without
relying on meaningful content (also called greeking). Replacing the actual
content with placeholder text allows designers to design the form of the
content before the content itself has been produced.

        The lorem ipsum text is typically a scrambled section of De
finibus bonorum et malorum, a 1st-century BC Latin text by Cicero, with
words altered, added, and removed to make it nonsensical, improper Latin.

        A variation of the ordinary lorem ipsum text has been used in
typesetting since the 1960s or earlier, when it was popularized by
advertisements for Letraset transfer sheets. It was introduced to the
information age in the mid-1980s by Aldus Corporation, which employed it
in graphics and word-processing templates for its desktop publishing
program PageMaker. Many popular word processors use this format as a
placeholder. Some examples are Pages or Microsoft Word.
    </h3>
    <button>
        <div>
            Click Me
        </div>
    </button>
</body>
</html>
```

Grid Areas

- Here we are looking at how to have the items we are placing on the grid take up more than one grid cell.

- The default for grid is that every item will take up one space on the grid.
- We can use several strategies to place the items on the grid, but we need to follow some rules:
    1. The item must be *self-contiguous* - that is, one item can take up multiple spaces, but they must be adjacent.
    2. The item can take up several rows, or columns, or an area (several rows and columns), but the shape is always a rectangle. (A square is just a particular type of rectangle.) This means it cannot form an L, or any other complex formation.