CS7026

The `div` Element and CSS for layout

CSS for Layout

- Having no layout whatsoever is only ok if all you want is one big column of content – highly unlikely.
- We need a way to divide up our content into different sections and position these sections on our page.
- Before we approach solving this problem, we need to be clear on the very important CSS `display` property.

The `display` property

- Every element has a default **`display`** value depending on what type of element it is.
- The default for most elements is usually **`inline`** or **`block`**.
- A **`block`** element is often called a block-level element.

Inline elements

- **Inline elements** are those which only occupy the space bounded by the tags defining the element, instead of breaking the flow of the content.

- The **a** element is a common inline element.

- **span** is another useful inline element. It allows you to wrap some text inside a paragraph **** like this **** without disrupting the flow of that paragraph.

Block Level Elements

- A **block element** starts on a new line and stretches out to the left and right as far as it can.

- Some of the block elements that we have already encountered include **`<p>`**, **`<h1>`** and **`<ul>`**.

The Mighty `<div>` Element

The most important block element for layout is the **`<div>`** element.

The **`<div>`** element is a container that divides your page into sections.

You can use it to group other elements in order to apply CSS to more than one element at a time.

Using CSS and the **`<div>`** tag, you can place elements exactly where you want them, without interrupting the flow of your document's structure.

The Tag

```
<div>
  <h3>Hi, welcome to the div tag.</h3>
  <p>
        <img src="mydiv.gif">
  </p>
  <p>All these elements are contained
  within a div</p>
</div>
```

- Now we have a way to divide up our content. But how do we go about specifying a different position on the page for each **`div`** element?

CSS - Class and ID Selectors

- Previously with CSS we looked solely at **HTML selectors** - those that represent an HTML tag.
- You can also define your own selectors in the form of **Class** and **ID** selectors.
- The benefit of this is that you can have the same HTML element, but present it differently depending on its class or ID.
- In the CSS, a class selector is a name preceded by a **full stop** (**.**) and an ID selector is a name preceded by a **hash character** (**#**).

Class and ID Selectors

- So the CSS might look something like:

```
#top {
  background-color: #ccc;
  padding: 1em
}
.intro {
  color: red;
  font-weight: bold;
}
```

Class and ID Selectors

- The HTML refers to the CSS by using the attributes **id** and **class**. It could look something like this:

```
<div id="top">
    <h1>Chocolate curry</h1>
    <p>This is my recipe for making curry
    purely with chocolate</p>
    <p class="intro">Mmm mm mmmmm</p>
</div>
```

Class and ID Selectors

- The difference between an ID and a class is that an ID should be used to identify ***one*** element, whereas a class can be used to identify ***more than one***.

- You can also apply a class or ID selector to a specific HTML element by simply stating the HTML selector first, so `p.jam {color:red;}` will only be applied to *paragraph* elements that have the class 'jam'.

- So now we have a way of dividing our page into sections (using the **`div`** element) and applying different CSS styles to each section by giving each **`div`** element a different **class** or **id**.

CSS Positioning

- How do we control where each **`div`** element appears on the page?

- We can use the **`position`** property which has a number of possible values:
  - **`static`**
  - **`relative`**
  - **`fixed`**
  - **`absolute`**

CSS Positioning - `static`

- **`static`** is the default value. An element with **`position: static;`** is not positioned in any special way.

- A static element is said to be *not positioned* and an element with its position set to anything else is said to be *positioned*.

```
.staticexample {
        position: static;
}
```

CSS Positioning - `relative`

- **`relative`** behaves the same as **`static`** unless you add some extra properties.

- Setting the **`top`**, **`right`**, **`bottom`**, and **`left`** properties of a relatively-positioned element will cause it to be adjusted away from its normal position.

- Other content will not be adjusted to fit into any gap left by the element.

CSS Positioning - **relative**

```
.relativeexample1 {
    position: relative;
}

.relativeexample2 {
    position: relative;
    top: -20px;
    left: 20px;
    background-color: grey;
    width: 500px;
}
```

CSS Positioning - **`fixed`**

- A **`fixed`** element is positioned relative to the viewport, which means it always stays in the same place even if the page is scrolled.

- As with **`relative`**, the **`top`**, **`right`**, **`bottom`**, and **`left`** properties are used.

- A **`fixed`** element does not leave a gap in the page where it would normally have been located.

CSS Positioning - **fixed**

```
.fixedexample {
    position: fixed;
    bottom: 0;
    right: 0;
    width: 200px;
    background-color: yellow;
}
```

CSS Positioning - `absolute`

- **`absolute`** behaves like **`fixed`** except relative to the nearest positioned ancestor instead of relative to the viewport.

- If an absolutely-positioned element has no positioned ancestors, it uses the document body, and still moves along with page scrolling.

- Remember, a "positioned" element is one whose position is anything except **`static`**.

CSS Positioning - **absolute**

```
.relativeexample {
    position: relative;
    width: 600px;
    height: 400px;
}

.absoluteexample {
    position: absolute;
    top: 120px;
    right: 0;
    width: 300px;
    height: 200px;
}
```

CSS Positioning - `z-index`

- The **`z-index`** determines which elements are drawn over others.
- Eg., if you have two elements that inhabit the same space, you need to specify which gets drawn and which is hidden.
- The one with the highest z-index number gets placed on top, while the one with the lowest gets placed on the bottom.

CSS Positioning - **z-index**

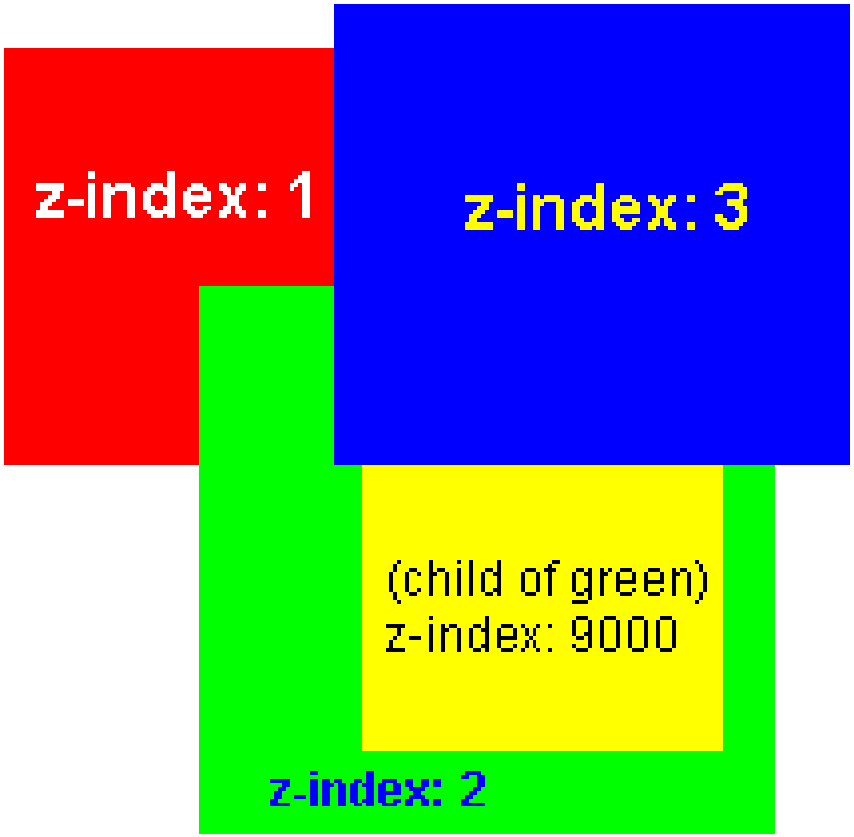
```
#reddiv{
      position:absolute;
      left:235px;
      top:110px;
      width:150px;
      height:150px;
      background-color:red;
      z-index:2
}
```

CSS Positioning - `z-index`

- ***This order is relative to the parent element.***

- Even if an element has a **`z-index`** of a million, if its parent is at the bottom of the **`z-index`**, it can't rise above it.

CSS Positioning - `z-index`



CSS Positioning - `visibility`

- **`visibility`** controls whether or not the element is drawn on the screen.
- values are **`visible`** and **`hidden`**, which are pretty much self-explanatory.
- Like all CSS values these can be dynamically controlled.

CSS Layout – Position Example

- This position stuff might make a little more sense in a practical example. Below is a realistic page layout.

```
.container {
    position: relative;
}
.navigation{
    position: absolute;
    left: 0px;
    width: 200px;
}
```

CSS Layout – Position Example

```
.content{
    /* position is static by default */
    margin-left: 200px;
}
.footer {
    position: fixed;
    bottom: 0;
    left: 0;
    height: 70px;
    width: 100%;
}
```

CSS Positioning

- Now you can position things on the page, to the exact pixel but this isn't a very flexible way to manage layout. We will improve on this in the following lectures.

- Please remember that people have monitors and browsers that are different sizes than the one you are currently using.