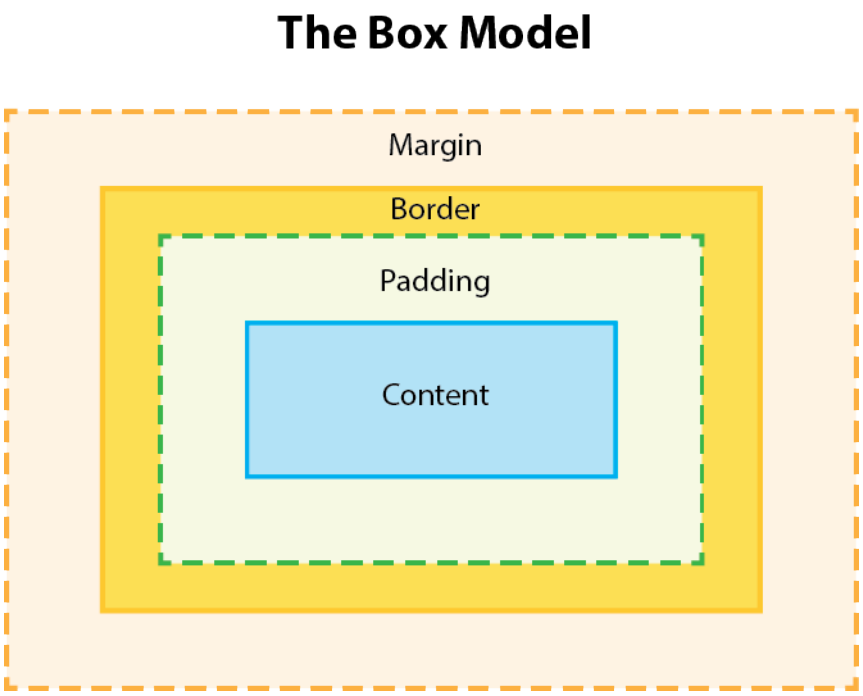
CSS Review
Layout & Pointers

If you are participating in this tutorial, it is assumed that at this point you have completed the introductory HTML & CSS track on Codecademy or some other overview assigned by the instructor. The purpose of this review is to address some of the most important aspects of CSS in more detail.

The Box Model



Content: The content of the box is an html element such as an <p>, <img>, , etc. Pretty much any element denoted by an html tag can be considered content.

Padding: padding is added to create space around the content, but inside the border. If you have applied a background color to the html element, the padding will inherit that color.

Border: A border goes around the padding and content. A border is typically used to add, well, a border, whether it be solid, dashed, or otherwise. The color and style of the border can be specified in the CSS.

Margin: The margin creates space around the border. The margin is always completely transparent and should be used to create space between separate html elements.

Box Model Example
See it live: http://codepen.io/cmealo/pen/tdDow

HTML:

```
<html>
<head>
</head>
<body>

 <img src="puppy.png"/>

 <p> This is a little baby paragraph</p>

</body>
</html>
```

CSS:

```
body{
  margin: 0px;
}

img {
   background-color: blue;
   padding: 50px;
   border: 10px solid red;
   margin: 25px;
}

p {
   background-color: pink;
   padding: 10px;
   border: 10px solid purple;
   margin: 0px;
}
```

***please note:** Browsers automatically add an 8px margin to the body which is why I manually set the body to have a zero margin.

Result:

This is a little baby paragraph

IDs and Classes

When I was first starting to code I was a little confused about the difference between IDs and Classes. I was told only to apply an ID to one thing but I noticed when coding that I could actually use an ID for multiple elements and the code still seemed to work. You do NOT want to do that.

IDs are UNIQUE.

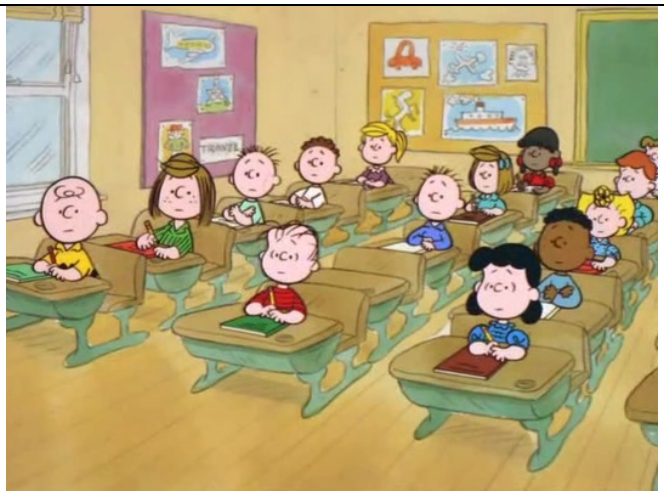
- Each page element can have only one ID.
- An ID can only belong to one element on a page.

Classes are NOT UNIQUE

- The same class can be used on multiple elements.
- An element can have multiple classes.

An element can have an ID and one or more classes.

This metaphor describes it best:

	
ID Like a student ID, an ID attribute is unique and should represent only one html element on a page. IDs can be used to identify and uniquely style a particular element on a page.	**Class** A class attribute is kind of like a class at school. Just as multiple students belong to a class, multiple HTML elements can have the same class attribute. And just as a student can attend a variety of different classes, so too can one HTML element have a number of different classes. Classes can be used to group and style multiple elements in the same way.

ID & Class Example

Live Example: http://codepen.io/cmealo/pen/wjcHF

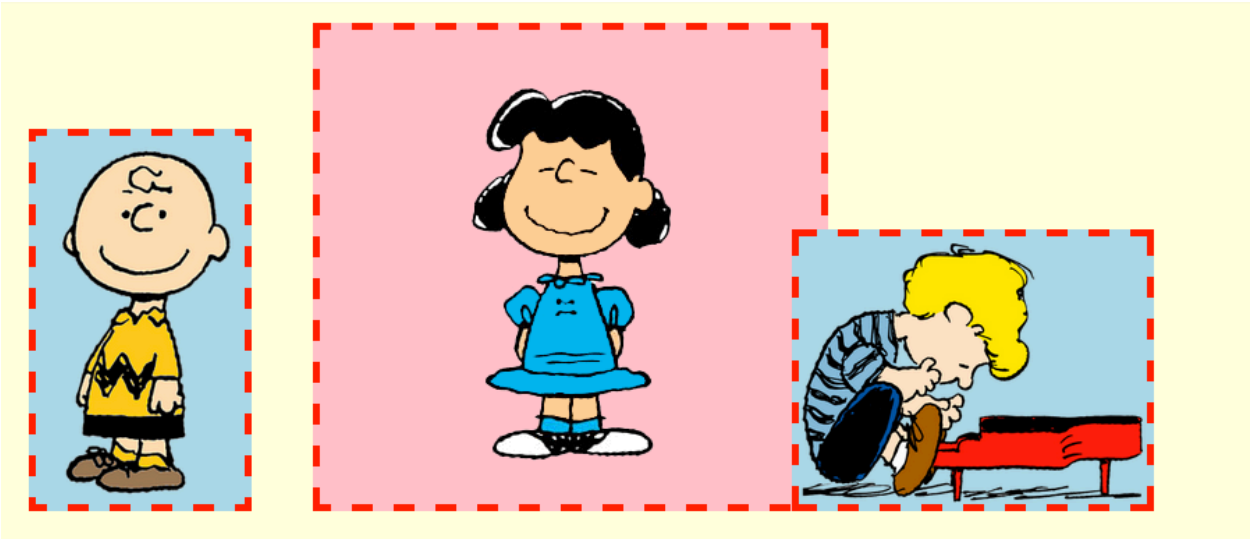
HTML:

```
<body>
   <div class="cartoon">

          <img id="charlie" class="peanuts boy"
         src="http://upload.wikimedia.org/wikipedia/en/2/22/Charlie_Brown.png"/>

          <img id="lucy" class="peanuts girl"
         src="http://upload.wikimedia.org/wikipedia/en/e/e9/Lucy_van_Pelt.png" caption="Lucy"/>

          <img id="schroeder" class="peanuts boy"
         src="http://upload.wikimedia.org/wikipedia/en/b/b7/Schroeder_Piano.png"/>

    </div>
</body>
```

CSS:

```
body{
  margin:0px;
}

.cartoon{
  background-color: lightyellow;
  padding: 20px;
}

.peanuts{
  border: 5px dashed red;
}

.girl{
  background-color: pink;
}

.boy{
  background-color: lightblue;
}

#charlie{
  margin-right: 40px;
}

#lucy{
  padding: 30px;
}

#schroeder{
  margin-left: -30px;
}
```

Result:



In the example above Charlie, Lucy, and Schroeder each have their own unique ID tag to identify them uniquely and add some unique styling. Charlie and Schroeder also have the class “boy” which makes their backgrounds blue; (*note this works for pngs with transparent backgrounds*) Lucy has class “girl” which is responsible for the pink background. All three of them have the class “peanuts” which gives them a red dashed border. The parent div has the class “cartoon” which has been styled with a yellow background.

It is important that IDs are only used for one element on a page. Sometimes you will notice IDs used to identify sections of a page because you can alter the URL to scroll down to a particular ID if you want to (ex scroll down to Charlie should scroll down the page until it gets to the item with the ID “charlie"). Your code will not pass validation if a particular ID is being used for more than one page element.

Please Note: To give an element multiple classes, all you have to do is add spaces between the names of the classes within the “” marks. See **example below:**

<img id="charlie" class="**peanuts boy**"
src="http://upload.wikimedia.org/wikipedia/en/2/22/Charlie_Brown.png"/>

Above, “peanuts” and “boy” are two different classes. This is indicated by the space that separates them where the class attribute is being defined.

CSS Layout Basics

Inline vs. Block Display

Most HTML elements naturally display as **block level elements** or as **inline elements**.

Inline Elements: take up as much width as necessary, and do not force line breaks.

Examples of inline elements:

-
- <a>
- <img>

The above are the most common inline elements you will come across.

Block Level Elements: take up full width available, and have line breaks before and after.

Examples of block elements:

-
- <h1>… <h6> (all headings)
- <p>
- , <ol>, <dl> (lists)
- , <dt>, <dd> (list items and definitions)
- <table>

Changing how an element is displayed:

The **display property** can be used to change inline elements to block level elements and vice versa. There is also an option for '**inline-block**' which displays an element as an inline-level block container. All display options: http://www.w3schools.com/cssref/pr_class_display.asp

Ex: Displaying a list item as an inline element instead of a block level element:

li {**display**: inline;}

Live Ex: http://www.w3schools.com/css/tryit.asp?filename=trycss_display_inline_list

Ex: Displaying a span as a block level element instead of an inline element:

span {**display**: block;}

Live Ex: http://www.w3schools.com/css/tryit.asp?filename=trycss_display_block

Floating & Clearing

Float

Floating an element takes it out of the normal flow and pushes it as far as it can go to the right or left within the parent element until it encounters another floating element or reaches the left or right bounds of the parent element.

Elements can be floated horizontally, which means that an element **can be floated left or right**; not up or down. Ex**: float: left;** or **float: right;**

- A floated element moves as far as possible to the left or right of the parent element.
- The elements after the floating element will flow around it.
- The elements before the floating element will not be affected.

If an image is floated to the right, a following text flows around it, to the left.

Clear the float

Elements that come after the floating element will flow around it, often becoming hidden behind floating elements and causing all sorts of layout issues. To avoid this, use the **clear** property.

- The clear property specifies which sides of an element other floating elements are not allowed.
- There are three options:
    - **clear: right;**
    - **clear: left;**
    - **clear: both;**

So if you have an element floating to the left, **clearing the following element left** will bring it down under the left floating element rather than becoming hidden behind the floating element.

If you have an element floating right, **clearing the following element right** will bring it down under that element to avoid any interference from right floating elements.

Typically, you will use clear: both; because if you have any element that comes after floating elements, whether they be floating left, right, or a combination, clearing both will clear all floating elements and bring the following element down below them to avoid any layout issues or interference.

More on Floating & Clearing: http://css-tricks.com/all-about-floats/

Float & Clear Example

Live Example: http://codepen.io/cmealo/pen/nyqif

HTML:

```
<body>
   <div class="left-float pink"></div>
   <div class="left-float yellow"></div>
   <div class="right-float blue"></div>

      <div id="wrapper">
         <div class="left-float pink"></div>
         <div class="left-float yellow"></div>
         <div class="left-float orange"></div>
        <div class="right-float blue"></div>
      </div>
</body>
```

CSS:

```
body{ margin: 0px; }

div{
  width: 33.33%;
  height: 80px;}

.left-float{ float: left; }

.right-float{  float: right;}

.pink{ background-color: pink; }

.yellow{ background-color: yellow; }

.blue{ background-color: lightblue; }

.orange{ background-color: orange;}

#wrapper{
  background-color: green;
  padding: 40px;
  width: 66.66%;
  clear: both;}

#wrapper div{
  width: 25%;
  height: 80px;}
```

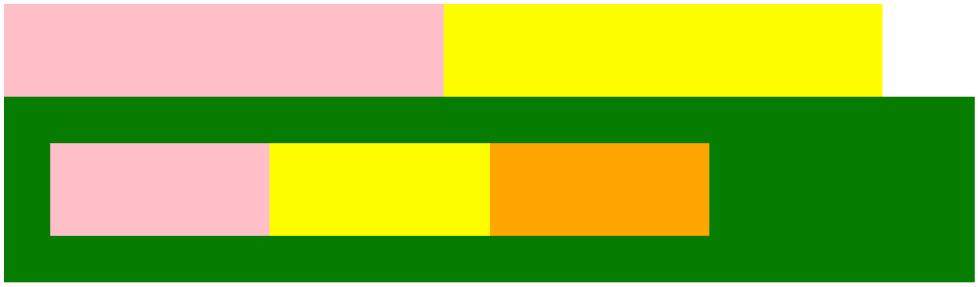
---

* **Note** #wrapper div{ } styles all divs that are children to the parent with id="wrapper"

Result:

Here all pink, yellow, and orange divs have been given a left **float**. Blue divs are floating right. The widths of the boxes have percentage values to ensure the full space of the parent element is being occupied. The green box (#wrapper) has to **clear** the first three divs because they are all floating, therefore have been taken out of the normal flow, without clearing the green box it will attempt to flow around the floating elements and will become partially hidden behind them.

What happens if we remove the right float on the blue boxes? (try on live example)
AKA we delete **.right-float{ float: right;}**
The result is pictured below:



What has happened to the blue boxes? Well, they are not floating like their sibilings, nor are they clearing, so they are ignoring the floating elements, which have been taken out of the default flow. The blue boxes are behaving according to how they would if there were no floating elements at all and are now hidden behind the pink boxes.

Note: you may have noticed that the width of #wrapper was set to 66.66% and may be wondering then why the green is extending past the yellow of the row above where all of the boxes are set to 33.33%. Well, padding gets tacked on to the width; so if we think about the **Box Model**, the total width of the content itself (the wrapper div) is 66.66% of the page width. So, the four boxes inside #wrapper together will be the same width as the combined width of the pink and yellow box in the row above. The padding however, is extending the space around #wrapper by an additional 40px on all sides. This shows how using a combination of percentage values and px values can at times make things a bit tricky.

Positioning

The position property is another way to position elements.
There are four different positioning methods: **Static**, **Absolute**, **Relative**, and **Fixed**.

Elements can be positioned using **Top**, **Bottom**, **Left**, and **Right** properties but the effect will vary depending on which of the four methods is being used.

Static Positioning

By default, html elements are positioned statically. **Static Positioning is automatic if no other value is specified**, and represents the **normal flow**; in most cases you want to stick to statically positioned elements unless there is a very good reason not to.

Fixed Positioning

Fixed positioned elements are removed from the normal flow. They **are fixed to specific pixel coordinates or percentage values within the viewing window** and will cover other elements even as you scroll down a page.

When to use: There is one particularly good case to use fixed positioning, and that is when creating a **navigation** bar that you would like to have remain at the top or bottom of your page even as the user scrolls down a long page.

Relative positioning

Relatively positioned elements are **positioned *relative* to their normal position in the flow**; in other words, *relative* to where the element would normally appear in the browser window if it were statically positioned. The content of relatively positioned elements can be moved and overlap other elements, but the reserved space for the element is still preserved in the normal flow.

When to use: The most **typical use for relative positioning is to create a container for absolutely positioned elements**. *Absolutely positioned elements are positioned relative to the first parent that has a position other than static*; so **containers for absolute elements need to either have relative or fixed** positioning. Relative is more useful for this task.

Absolute Positioning

Like relative positioning absolute positioning positions an element relatively, but not to where the element would normally exist in the flow. Rather, an absolutely positioned element is **positioned relative to the first parent element that has a position other than static**. If no such element is found, the containing block (parent element) is <html>.

Absolutely positioned elements are taken out of the flow and assigned a specific location; they can overlap other elements in a document.

When to use: The most typical use for absolute positioning is inside of a relatively positioned container. In cases where you want to have several overlapping images hiding and toggling, or when there are a number of objects that need to be placed relative to the boundaries of a parent container, absolute positioning can be useful.

Positioning & Layout Example

Live Example: http://codepen.io/cmealo/pen/zEnrK

HTML:

```
<body>
<div class="fixed navbar">
        <ul id="primary-nav">
    <li>Home</li>
    <li>About</li>
    <li>More</li>
  </ul>
 </div>

 <div class="content-wrapper">
        <div class="float-left left-column">

                <div class="relative" id="relative-container">

                        <div class="absolute" id="absolute-1"></div>

                        <div class="absolute" id="absolute-2"></div>

                        <div class="absolute" id="absolute-3"></div>


                </div> <!-- end of relative container-->

        </div>

        <div class="float-right right-column">

                <div class="static">
                        <h1> Heading </h1>
                </div>

                <div class="static">
                <p>Lorem ipsum dolor sit amet...</p>

                 <p>Ne liber oratio soleat pro...  </p>

                <p>Mediocrem constituam in pri...</p>

         </div>
      </div>
    </div>
  </body>
```

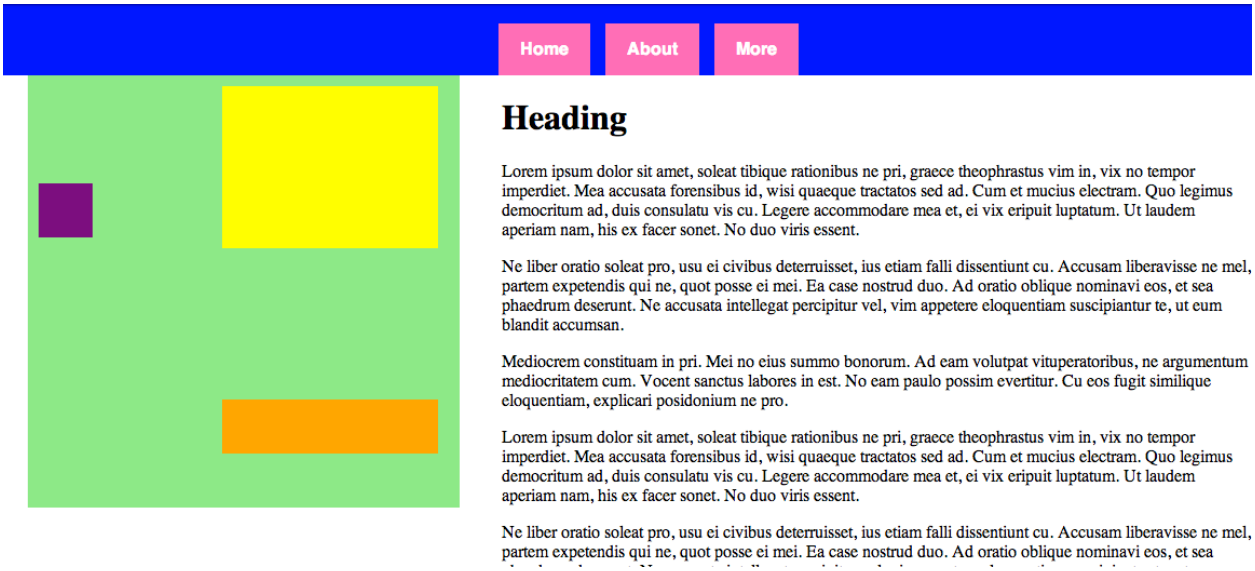
CSS:

```
body{
  margin: 0px; }

.navbar{
  width: 100%;
  margin: 0 auto;
  background-color:blue;
  padding-top: 20px;
  font-family: sans-serif;
  font-weight: bold;
  color: white; }

.content-wrapper{
  width: 100%; }

#primary-nav {
  margin: 0 auto;
  list-style:none;
  text-align:center; }

#primary-nav li{
  display: inline-block;
  background-color: hotpink;
  margin: 0 5px;
  padding: 15px 20px; }

.float-right{
  float: right; }

.right-column{
  width: 60%; }

.float-left{
  float: left; }

.left-column{
  width: 40%; }

.fixed{
  position: fixed;
  z-index: 99999; <!-- Note: a high z index will bring it forward on the Z axis, otherwise
                       overlapped by the relative box. See what happens on live example-->
}
```

```
.static{
  position: static;}

.relative{
  position: relative; }

.absolute{
  position: absolute; }

.navbar.fixed{
  position: fixed;
  top: 0px; }

.content-wrapper{
  margin-top: 68px; }

#relative-container{
  width: 400px;
  height: 400px;
  background-color: lightgreen;
  left:25px; }

#absolute-1{
 width: 50px;
  height: 50px;
  background-color: purple;
  top: 100px;
  left: 10px; }

#absolute-2{
 width: 200px;
  height: 50px;
  background-color: orange;
  bottom: 50px;
  right: 20px; }

#absolute-3{
 width: 200px;
  height: 150px;
  background-color: yellow;
  top: 10px;
  right: 20px; }
```

Note: This is easier to understand and play around with on the live example:
http://codepen.io/cmealo/pen/zEnrK

Results:



In the above there is a left floating column and a right floating column.

#relative-container is the green div in the left column. It has been positioned as relative by way of a class attribute called "relative" and has a left indent ***relative*** to where it would normally fall in the flow, which is specified under the styling for its **ID** (**left:25px;**).

```
<div class="relative" id="relative-container">

.relative{
 position: relative; }

#relative-container{
 width: 400px;
 height: 400px;
 background-color: lightgreen;
 left:25px; }
```

You can also see in the above picture that the small absolutely positioned boxes: **#absolute-1**, **#absolute-2**, and **#absolute-3** have been **absolutely positioned** in relation to the bounds of their relatively positioned parent div (#relative-container).

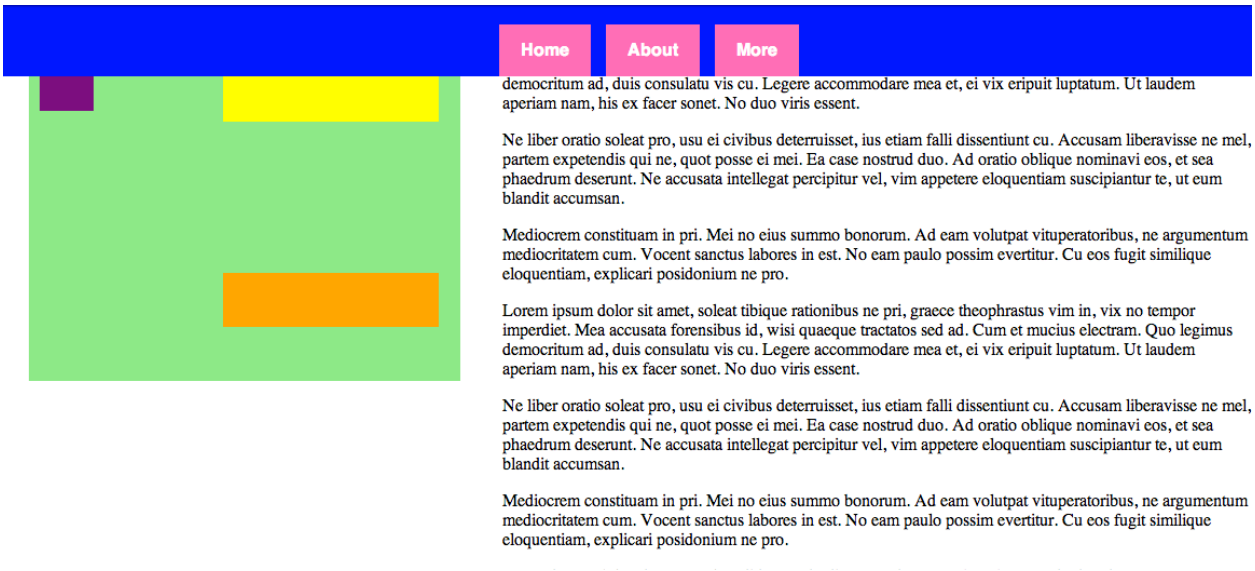
The div with the class "navbar" also has a class named "fixed". The "fixed" class has been styled with a fixed position and a **z-index** of 999999, which is just a high number intended to bring all fixed elements to 'the front' on the z-axis so that no other elements can overlap them.

(see on next page)

```
.fixed{
  position: fixed;
  z-index: 99999; <!-- Note: a high z index will bring it forward on the Z axis, otherwise
                          overlapped by the relative box. See what happens on live example-->}

.navbar.fixed{
  position: fixed;
  top: 0px; }
```

.navbar.fixed selects all items that have both the "navbar" class and the "fixed" class so that we can style any elements with both classes; this is a **complex CSS selector**.

Now we can style the navbar with **top:0px;** which will anchor it in a fixed position to the top of the browser window. **As we scroll down the navbar should come with us:**



See what happens **when z-index is removed**; the relative div comes to the top of the stack: