# Three Pillars To Write Good HTML and CSS

1. Responsive Design

* Fluid Layouts
* Responsive Images
* Correct Units
* Desktop First vs Mobile First

1. Maintainable and Scalable Code
   * Clean
   * Easy to Understand
   * Growth
   * Resusable
   * How to organize files
   * How to name classes
   * How to structure HTML
2. Web Performance
   * Less HTTP requests
   * Less Code
   * Compress code
   * Use a CSS Preprocessor (LESS/SASS)
   * Less Images
   * Compress Images

# How CSS Works Behind The Scenes

1. The browser loads the HTML
2. Then parses the HTML and builds the Document Object Model (DOM)
   1. Describes the parent, children and sibling elements.
3. The browser loads the CSS
   1. Parses the CSS declarations (cascade) the processes the final CSS values
4. The final CSS is also stored in a tree like structure similar to the DOM, called CSS Object Model (CSSOM)
5. With the CSSOM and the DOM, we now have the rendering tree which is the website rendering/visual formatting

# How CSS Is Parsed, Part 1: The Cascade and Specificity

Consists of a 'selector' and a declaration block:

.my-class {

color: blue;

text-align: center;

font-size: 20px;

}

### The Cascade (The 'C' in CSS)

The cascade is the process of combining different stylesheets and resolving conflicts between different CSS rules and declarations, when more than one rule applies to a certain element.

CSS can come from several sources:

* The developer/author
* User
* Default browser declaration

The cascade combines all of the declarations from these different sources.

The cascade resolves the importance, specificity, and source order to determine which takes precedence.

**Order of Importance:**

* User !important declarations
* Author !important declarations
* Author declarations
* User declarations
* Default browser declarations

**Order of Specificity:**

* Inline styles
* Ids
* Classes, pseudo-classes, attributes
* Elements, pseudo-elements

With Source Order:

* The last declaration in the code will override all other declarations and will be applied.

CSS declarations mare with !important has the highest priority.

Only use '!important' as a last resource. More maintainable code.

Inline styles will always have priority over styles in external stylesheets.

To be really specific concerning a style, use an 'ID'.

Rely more on specificity than the order of selectors.

But, rely on order when using 3rd party stylesheets – always put your author stylesheet last.

Visit codepen.io to test css for selector specificity.

# How CSS Is Parsed, Part 2: Value Processing

Padding initial value is 0px.

The default font size of the browser is 16px. A specified font size of '1.5rem' equals to 24px (16 \* 1.5)

Font sizes are inherited by it's parent. A font size of 150% will 1.5 times the size of it's parent.

The same is true with length. A length of 10% is relative to it's parent.

'*Em*' size are relative to the font-size of the parent or current element. '*Rem*' uses the root font-size (16px).

Lengths depend on the font size.

*VH* and *VW* are % of the viewport height and width.

# Inheritance

* Inheritance passes the values for some specific properties from parents to children and allows developers to write less code – **more maintainable code**
* Properties related to text are inherited: font-family, font-size, color, etc;
* Other properties like margins or padding are not inherited
* Inheritance of a property only works if no one declares a value for that property
* The '*inheri*t' keyword forces inheritance on a certain property
* The '*initial'* keyword resets a property to its initial value

# Convert REM into pixels

* Convert '*px*' in to '*rem*'. At 16px, 1 px = 0.0625rem
* CSS to enter: body { font-size: 62.5%; }
* 1rem is now equivalent to 10px. 1.6rem = 16px

# How CSS Renders a Website: The Visual Formatting Model

The VSM is an algorithm that calculates boxes and determines the layout of these boxes, for each element in the render tree, in order to determine the final layout of the page. This is **one of the fundemental models** in CSS.

The algorithm takes into account the:

* **Dimensions of boxes**: the box model;
* **Box type**: inline, block and inline-block;
* **Positioning scheme:** floats and positioning;
* **Stacking contexts**;
* Other elements in the render tree;
* Viewport size, dimensions of images, etc.

**The Box Model**

The box model is the most fundemental and important to master CSS. According to the box model, each and every element can be seen as a rectangular box. Each ‘box’ can have a width/ height, padding, border, and margin. Broken down:

* **Content**: text, images, etc. Also the width and height of the box;
* **Padding**: transparent area around the content; inside of the box;
* **Border**: goes around the padding and the content;
* **Margin**: space between boxes;
* **Fill area**: area that gets filled with background color or background image. This gets

applied to the entire box, including the content and padding; but not the margin.

**Heights and Widths**

By default:

Total width = right border + right padding + specified width + left padding + left border

Total heigth = top border + top padding + specified height + bottom padding + bottom border

**Ex**: height = 0 + 20px + 100 px + 20px + 0 = 140px

This doesn’t sound very practical. For this use **box-sizing: border-box**. This defines the height and width of the box, according to the specified dimensions; and not including the padding and border:

Ex: height = 0 + 20px + 100px + 20px + 0px = 100px

This makes things so much easier.

**Box Types: Inline, Block-Level and Inline-Block**

Block-level:

* Elements formatted visually as blocks
* 100% of parent’s width and create line breaks, after and before it
* Vertically, one after another

By default, all html elements are ‘block’ layouts in CSS. These all display the ‘block’ layout in CSS:

* display: block
* display: flex
* display: list-item
* display: table

Inline Boxes are basically the opposite of block-level boxes:

* Content is distributed in lines
* Only occupies the space the content actually needs
* No line breaks
* No heights and widths can be used
* Paddings and margins are only horizontal. Not vertical

Ex: display: inline

This of course has its limitations. To overcome them, there’s an **inline-block** box. These are inline boxes, but work as an block-level box on the inside.:

* A mix of block and inline
* Occupies only its content space
* No line-breaks
* Ex: display: inline-block

**Positioning Schemes**

Normal Flow is what happens to an element when you don’t do anything to it:

* Default positiong scheme;
* **NOT** floated;
* **NOT** absolutely positioned;
* Elements laid out according to their source order
* **Default**:
* position: relative;

Floats:

* Element is removed from the normal flow and moved to the left or right as far as possible;

until it touches the edge of its containing box or another floated element.

* Text and inline elements will wrap around the floated element;
* The container will not adjust its height to the element. This can be problematic. The usual

solution to this is to use ‘clear’ fixes (clear: both;).

* Ex:, float: left; , float: right;

Absolute:

* Element is removed from the normal flow
* No impact on surrounding elements or content;
* We use *top, bottom, left* and *right* to offset the element from its relatively positioned

container.

* Ex: position: absolute, position: fixed

An absolutely positioned element can overlap other elements occupying the same space. CSS solves by using ‘*Stacking Contexts*’. *Stacking Contexts* determing which order elements are rendered on the page.

The most widely known stacking context is ‘*z-index*’. The element with the highest z-index appears on top; the lowest at the bottom.