Agenda

- SCSS/SASS
- Introduction to Basic SCSS
- DRY CSS
- CSS Architecture Paradigms
  - Atomic CSS
  - OOCSS
  - BEM
  - SMACSS
  - ITCSS
- Understanding different paradigms through example
- Hands on building a responsive CSS Grid(bootstrap like)
- Hands on architecturing a stylebase for future projects with above paradigms.

WIFI : OSI2019

Password: 2019@osiws

Speakers - Introduction

Bhavan Kuchibhotla

senior Front End Engineer, pushEngage, 6+ years of Front UI Focused End Engineer. ex-Freshworks, ex-cleartax, ex-oyorooms

Anjanish Kumar

Tech Lead, pushEngage
7+ years of Full Stack Engineering

Mrityunjay Kumar

Front End Engineer, pushEngage, 3+ years Full Stack experience, Front End Enthusiast

SCSS/SASS

- SASS/SCSS (Syntactically Awesome Style Sheets / Sassy CSS) is a CSS extension language and compiler designed by Hampton Catlin & developed by Natalie Weizenbaum back in RoR days (2006 nov28) initially written in Ruby Language.
- Its a Preprocessing Scripting language that is interpreted/compiled sassScript into CSS files.
- Two Formats:
  - **Sass** - file extension **.sass** - syntax similar to haml templating language, i.e intendation based, older format. More familiar to RoR devs.
  - **Scss** - file extension **.scss** - syntax similar to Natural Css, i.e., block based, more familar format to all devs.

History

- Started as a Ruby Gem initially. Ruby Gem Implementation is no more maintained due to lack of resources and has officially reached its end of life few months ago(March’ 19).
- Later due to compilation time concerns,Hampton Catlin wrote the sass compiler in C++ as Libsass and also available in dart implementation (dartSass).
- Major languages currently have wrappers to libsass.
    - sass (npm module for javascript)
    - node-sass (node wrapper on top of libsass)
    - Phamlp - unofficial PHP Implemetation
    - Jsass - unofficial Java implemtation
- We are going to use sass npm module for hands-on session today.

SCSS Setup for Hands on Coding today

- Options we have: node-sass module, vs-code scss extension(local scss compiler),
- Incase you can’t / don’t want to setup on your local machine,try using codepen.io it has scss preprocessor available, but no support for file system.
- Using npm:
    - npm install node-sass, and npm install nodemon

SASS Introduction

- Variables
- Operators
- Interpolation
- Lists and Maps
- Flow Control Statements
- Nesting
- Partials
- Mixins
- Functions
- Inheritance (@extend)
- Q&A regarding the Language basics

SASS Variables

- Variables are declared with a $ before the variable name, once defined, they can be used anywhere that scss file is used/imported.

```
$primary: #456bbc;
$secondary: #FFA940;
$white: #fff;

.btn{
        padding: 8px 20px;
        color: $white;
        display: inline-block;
        text-align: center;
        &-primary {
        background-color: $primary;
        }
        &-secondary {
        background-color: $secondary;
        }
}
```

```
.btn {
  padding: 8px 20px;
  color: #fff;
  display: inline-block;
  text-align: center;
}
.btn-primary {
  background-color: #456bbc;
}
.btn-secondary {
  background-color: #FFA940;
}
```

Operators

- Equality Operators (==, !=)
- Relational Operators (<, <=, >, >=)
- Arithmetic Operators (+, -, *, /, %)
- Unary Operator (-, +)
- Logical Operators (and, or, not)

Equality Operators

- Equals to (==) and Not Equals To (!=)
- The equality operators returns whether or not two values are the same.

```
// Number
16px == 16 ; // false
96px == 1in ; // true
// String
“Helvetica” == Helvetica;  // true
“Helvetica” == Arial;  // false
// Colors
hsl(34, 35%, 92.1%) == #f2ece4; // true
rgba(179,115,153,0.5) != rgba(179,115,153,0.8);
```

```
// List
(5px, 7px, 10px) == (5px, 7px, 10px) ; // true
(5px, 7px, 10px) == (5px  7px  10px) ; // false
(5px, 7px, 10px) == [5px, 7px, 10px] ; // false
// Map
$mapA: (“a”: 1, “b”: 2)
$mapB: (“a”: 1, “b”: 2)
$mapC: (“a”: 2, “b”: 1)
$mapA == $mapB ; // true
$mapA == $mapC ; // false
```

Relational Operators

- **<**, **<=** , **>**, and **>=**
- Relational operators determine whether numbers are larger or smaller than one another.
- Automatic conversion between compatible units.

```
50 > 25 // true
100px < 200px // true
96px >= 1in // true
1000ms <= 1s // true
```

```
100 > 50px // true
100px > 100s
// Error: incompitable units px and s
```

Arithmetic Operators

- **+, - , *,** / and **%**
- Arithmetic operations can only be performed on same unit Values/ compatible values
  - Compatible values in, cm, px are compatible values., vice versa s, px are not compatible.
  - Unitless numbers can perform arithmetic operations with any values.
  - + Behaves as concatenation in case of string values, otherwise mathematical addition operator.

```
1px + 1in = 97px
1px + 1cm = 38.79px
1cm + 1in = 3.54cm
1px + 1em = Error: Incompatible units em and px.
$value: 20; $unit: px
$value + $unit = 20px
```

```
6 / 2 = 6/2
(6 / 2) = 3
(6px / 2) = 3px
(6px / 2px) = 3
(5px % 2) = (5 % 2px) = (5px % 2px) = 1px
```

Unary Operators

- **+**, **-** and / can also be used as unary operator.
- Unary operator takes only one value a operand.
- Always write spaces on both sides of - when subtracting.

```
a-1 = a-1
2px-1px = 1px
5px-3 = 2px
5 - 3 = 2
5 -3 = 5 -3
1 -2 3 = 1 -2 3
1 - 2 3 = -1 3
```

```
$number: 2
1 -$number 3 = -1 3

/ 15px = /15px
```

Logical Operators

- **and, or, not**
- Sass uses words rather than symbols for its boolean operators.
- The value *null* and *false* are falsy anything else is consider truthy.

not true = false

not false = true

not null = true

true and true = true

true and false = false

false or true = true

false or false = false

Interpolation

- string/numeric/boolean values are interpolable in style declaration, properties, property values

```
@mixin my-icon($name, $vAlign: top, $hAlign:left) {
  position: absolute;
  #{$vAlign}: 0;
  #{$hAlign}: 0;
  background-image: url("/icons/#{$name}.svg");
}

.left-icon {
  @include my-icon("left", top, left);
}

.close-icon {
  @include my-icon("close", bottom, right);
}
```

```
.left-icon {
  position: absolute;
  top: 0;
  left: 0;
  background-image: url("/icons/left.svg");
}

.close-icon {
  position: absolute;
  bottom: 0;
  right: 0;
  background-image: url("/icons/close.svg");
}
```

List

- List is used to store sequence of other values same as array.
- Lists can be comma/space separated.
- List is Sass is immutable.

```
$colors: (red, blue,green)

nth($colors, 1) // red
nth($colors, -1) // green

append($colors, yellow) // (red, blue, green, yellow)
append([10px 20px], 30px) // [10px 20px 30px]


index(1px solid red, solid) // 2
index(1px solid red, dashed) // null
```

```
set-nth($colors, 1, yellow) // (yellow, blue, green)

is-bracketed($colors) // false

join($colors, yellow) // (red, blue, green, yellow)

length($colors) // 3
```

List

```scss
$sizes: 40px, 50px, 80px;

@each $size in $sizes {
  .icon-#{$size} {
    font-size: $size;
    height: $size;
    width: $size;
    background: #eee;
  }
}
```

$\rightarrow$

```css
.icon-40px {
  font-size: 40px;
  height: 40px;
  width: 40px;
  background: #eee;
}

.icon-50px {
  font-size: 50px;
  height: 50px;
  width: 50px;
  background: #eee;
}

.icon-80px {
  font-size: 80px;
  height: 80px;
  width: 80px;
  background: #eee;
}
```

Map

- Map in Sass is used to store key value pairs similar to object.
- Maps allow any Sass values to be used as their keys.

```
$font-weights: ("regular": 400, "medium": 500, "bold": 700);

map-get($font-weights, "medium"); // 500
map-get($font-weights, "extra-bold"); // null

map-has-key($font-weights, "regular"); // true

map-keys($font-weights); // (“regular”, “medium”, “bold”)
map-values($font-weights); // (400, 500, 700)

map-remove($font-weights, "regular", “medium”); // (“bold”: 700)
map-merge((“regular”: 400),(“medium”: 500)); // (“regular”: 400, “medium”: 500)
```

Maps

```scss
$colors: (
        light: #888,
        dark: #000,
        regular: #333
)

@each $key, $color in $colors {
  .text-#{$key} {
        color: $color;
  }

}
```

→

```css
.text-light {
  color: #888;
}

.text-dark {
  color: #000;
}

.text-regular {
  color: #333;
}
```

Flow Control

- Similar to most other languages, sass also has flow control
- @if and if
- @each (~ forEach flow control in javascript) to apply on lists/maps
- @for and @while(~ for and while/do-while flow control) to loop through values/conditions

Loops - FOR(@for)

- @for loop takes a iterator variable from some number/variable to(exclude)/through(include) another number/variable

```
//For Loop
$color: #456bbc;
$border-max: 5;
@for $i from 1 through $border-max {
       .border-#{$i} {
         border: #{$i}px solid #{$color};
       }
 }
```

→

```
.border-1 {
  border: 1px solid #456bbc;
}
.border-2 {
  border: 2px solid #456bbc;
}
.border-3 {
  border: 3px solid #456bbc;
}
.border-4 {
  border: 4px solid #456bbc;
}
.border-5 {
  border: 5px solid #456bbc;
}
```

Loops - While(@while)

- @while loop takes a condition and continues looping as long as that condition is true. We typically update the variable in condition on each iteration to prevent infinite loop

```
// while Loop
$i: 0;
$size: 5;
$color: #456bbc;
@while ($i <= $size) {
  .border-#{$i} {
      border: #{$i}px solid #{$color};
  }
  $i: $i + 1;
}
```

```
.border-0 {
  border: 0px solid #456bbc;
}
.border-1 {
  border: 1px solid #456bbc;
}
.border-2 {
  border: 2px solid #456bbc;
}
.border-3 {
  border: 3px solid #456bbc;
}
.border-4 {
  border: 4px solid #456bbc;
}
.border-5 {
  border: 5px solid #456bbc;
}
```

Loops - EACH(@each)

- @while loop takes a condition and continues looping as long as that condition is true. We typically update the variable in condition on each iteration to prevent infinite loop.

```
//Each Loop

$colors:  red , blue, yellow;

$gap: 8px;


@each $color in $colors {

  .btn-#{$color}{

          background-color: #{$color};

  }

}
```

```
.btn-red {
  background-color: red;
}

.btn-blue {
  background-color: blue;
}

.btn-yellow {
  background-color: yellow;
}
```

Nesting in SCSS/SASS

- You can write one style rules inside another style rule
- It has capability to generate hierarchical CSS
- Most Popular feature of Sass

```
ul.nav {
        padding:0;
        li {
                list-style: none;
                padding: 0 16px;
                color: #456bbc;

                &:hover {color: blue;}
        }
}
```

→

```
ul.nav {
  padding: 0;
}
ul.nav li {
  list-style: none;
  padding: 0 16px;
  color: #456bbc;
}
ul.nav li:hover {
  color: blue;
}
```

SASS Partials

- You don't have to write all your style in a single file.
- We can use @import to import files to another file to compile to CSS
- @use and @forward are a very latest features (released few weeks ago). It has namespacing, data privatisation ., etc.

```
//_typography.scss
$primary: #456bbc;
.text-bold{
        font-weight: 700;
}
.text-light {
        font-weight: 300;
}
//styles.scss
@import “./typography”;

p { color: $primary; }
```

→

```
.text-bold {
  font-weight: 700;
}

.text-light {
  font-weight: 300;
}
p {
  color: #456bbc;
}
```

SASS - Mixins

- Mixins are a great way to provide reusability in your code;
- Mixins are meant to return a block of styles.
- Parameters can be passed in to use conditional generation of style values.

```
//definition
@mixin btn() {
        background-color: #999;
        &:hover {
                background-color: #aaa;
        }
}
//usage
.button {
        @include btn();
}
```

→

```
.button {
  background-color: #999;
}
.button:hover {
  background-color: #aaa;
}
```

Mixins Examples

```
//definition
@mixin btn($size: “default”) {
    background-color: #999;
    @if $size == “small” {
        padding: 4px 10px;
        font-size: 12px;
    } @else if $size == “default” {
        padding: 8px 18px;
        font-size: 14px;
     } @else if $size == “large” {
        padding: 12px 24px;
        font-size: 16px;
     }
}
```

→

```
//usage
.button-small {
   @include btn(“small”);
}
.button-large {
   @include btn(“large”);
}
.button {
   @include btn();
    //no value for $size passed
}
```

→

```
.button-small {
  background-color: #999;
  padding: 4px 10px;
  font-size: 12px;
}

.button-large {
  background-color: #999;
  padding: 12px 24px;
  font-size: 16px;
}

.button {
  background-color: #999;
  padding: 8px 18px;
  font-size: 14px;
}
```

SASS Functions(@function)

- Write complex operations and reuse through out stylesheet.
- Functions can accept parameters,
- Note: Mixins are meant to return a block of css, while functions are to return any specific value we choose to use
- Examples, do some mathematical operations, check for existing variables in config etc.,
- @function functionName($vars){

```
    ……….

    @return somevalue;

}
```

SASS Functions(@function)

```
//List of colors
$colors:(
 primary: #456bbc,
 text: #333,
 shadow: rgba(0,0,0,0.2)
);
//definition
@function getColor($color){
          @if map-get($colors,$color) {
              @return (map-get($colors,$color));
            } @else {
              @warn 'There is no item "#{$color}" in this
          list; choose one of: #{$colors}';
              @return null;
          }
}
.box {
  color: getColor('text');
  background-color: getColor(‘primary’);
  box-shadow: 0px 0px 2px 2px getColor('shadows');
}
```

```
.box {
  color: #333;
  background-color: #456bbc;
  box-shadow: 0px 0px 2px 2px
rgba(0, 0, 0, 0.2);
}
```

Inheritance (@extend)

- We can inherit styles of class(.class) or placeholder(%placeholder) into another style declaration
- Mixins deffer here from @extend as mixins just return a block of styles, @extend will group all classes, style declarations together.

```
//definition

.btn {

   background-color: #999;

   &:hover {

      background-color: #aaa;

    }

}

%block {

  width: 100%;

}
```

```
//usage

.cta {

  @extend .btn;

  @extend %block;

}
```

```
.btn, .cta {
  background-color: #999;
}
.btn:hover, .cta:hover {
  background-color: #aaa;
}

.cta {
  width: 100%;
}
```

CSS Code architecture Paradigms

Common problems we face with CSS on a day-to-day basic

- Dead code: css code that our application won’t use anymore is still being served to the app.
- Specificity issues: Since CSS is global scoped, we have to make use of selector specificity frequently to achieve what we need, sometimes/most of the times by writing !important or overriding existing styling
- It eventually becomes impossible to maintain a style base, keep it readable, easy to debug when requirements change
- Developer Experience is dragged to hell by writing CSS without organizing it or following some proven workable ways of organization

CSS Code architecture Paradigms

Some brilliant engineers came up with few paradigms of organizing stylebase. Each serve different purposes. Based on our need, we can pick any of them and live longer life peacefully.

A Few of those paradigms:

- Dry CSS,
- Atomic CSS,
- OOCSS,
- BEM
- SMACSS
- ITCSS

All these paradigms try to solve similar problems we face coding UI with different approaches and rules

DRY (Do not Repear Yourself) - CSS

- Principles
  - Don’t repeat yourself : Never repeat a style/property definition if you can avoid it
  - Group Selectors with Shared properties rather than defining them separately
  - Individual selectors are rarely written and only used as exception
  - Name Style groups logically
  - Add selectors to various groups
  - Dry is mostly a software principle than a methodology
  - There are other methodologies which achieve this principle

```
//No DRY Principle followed
.btn {
        text-align:center;
        Display:inline-block;
}
.card {
        ....
        text-align:center;
        ....
}
```

```
//DRY way
.btn, .card, (...other selectors
those have text-align center) {
        text-align:center;
}
.card {
        ....
        ....
}
```

DRY (Do not Repeat Yourself) - CSS

- Pros:
  - You would never repeat any style/property
  - Reduces number of CSS lines generated thus reducing css file size
  - All stylistically matching selectors are grouped
  - Greatly useful in cases where html generation is out of our control., ex: third-party widgets, wordpress plugins etc.,
  - We can make use of Sass @extend to achieve this easily.
- Cons
  - Not very readable, difficult at first to figure out and debug
  - To change styles, we cut the selectors in css, and add them to other style groups

Atomic Css

- There are multiple variant of atomic css
- Initial Variant of ACSS specifies write your css classnames to look like functions like(BG(blue), Mt(10px) etc). There are tools which parse the html and generate css with those specific instructions., ex: Atomizer
- Another Variant of Atomic Css suggests to Create a class selector for every repeating css declaration.(ex: mt-10, mb-10, text-bold, text-light, col-6, col-12 etc.,)
- Considers each style declaration as a style atom and any ui can be made with a group of atoms.
    - Ex: `<div class="D(b) Va(t) Fz(20px)">Hello World!</div`
    - Atomizer tool would generate css styles like these

```
.D(b) {display: block;}

.Va(t){vertical-align: top;}

.Fz(20px) {font-size: 20px;}
```

Atomic Css

Utilities can be generated using sass

```
@for $i from 1 through 5 {
        .mt-#{$i}{
                margin-top : $i*10px;
        }
}
```

- Pros:
  - follows DRY principles.So, lesser number of lines of css over time
  - Can use preprocessors to generate a lot of repeating styles at scale easily with logic extens.,,
  - Atoms are extensible , flexible. All we need is to add more variations of style
  - Sass can be used with caution to scale the utilities, styles as needed.
  - We can make use of sass @for and loops to generate repeating styles
- Cons:
  - Html classes turn very verbose(subjective opinion*).
  - Some say the classes are not semantic. Almost a inline style replacement
  - Can become dead code if not used with caution.
  - As a paradigm it makes sense, but using atomizer is almost like a inline styling.

OOCSS- not really Object oriented technically

- Encourages us to think of a page as group of objects each with its styling to ensure reusability. OOCSS also ensures DRYness.
- Principles:
  - Separation of Structure from Skin, group style variations into structural and skin based classes and use multiple classes and variations in html
  - Declaration blocks are applied to elements using single-class selectors to avoid specificity issues
  - OOCSS involves identifying objects on a page and separating their structural and visual CSS styles into two declaration blocks. These blocks can then be reused by different elements, and changes need only be made in one place, leading to better consistency

  - Logically separate styles which define the structure/position(height, width, positioning, floats, margins, overflows.,etc.,) of a Object and skin of object(color, bg-color, shadows etc.,)

  - Styles declaration are grouped depending on structure (large, small, fixed, sticky etc) and skins(primary, secondary, light, dark, etc)

- Differences from ACSS
  - OOCSS recommends grouping based on structure and skin,containers
  - ACSS recommends each style variation to be a seperate group

OOCSS- cont.,

- Pros:
  - Almost follows DRY principle with a few more lines of styles.
  - Html has control over how the element looks
  - Great paradigm for writing styles from scratch / writing styles for components
  - Can use preprocessors to avoid specificity problem generally caused, by extending structure and skin to global level style and using that global level class in html. By writing semantic/required classnames in html, but extending desired styles from code objects.
- Cons
  - Styles of elements inside an object can’t be reused if not written with caution.
  - Lack of stricter rules result in different interpretations of the paradigm
  - Not very useful for layout level styling, in that case it almost resembles ACSS
  - Styles inside an object aren’t reusable in cases of different html structure

BEM - Block Element Modifier

- BEM is just a naming convention to tackle bugs those come due global nature of css
- When you have specific style declarations like .header > .nav li a+span etc., we can't really reuse the style we used in span element here
- BEM suggests to have every styling with same level specificity.
- B- Block
    - Any style block (header, card, loginForm etc.,)
    - Bem Suggests to use a very clear naming for the block
- E- Element
    - Any element or group of elements in a Block which need a specific styling(header__logo, loginForm__input, loginForm__label etc.,)
    - Elements are generally separated by __ from block.
- Modifier
    - Any different variant or Block or Element (header--sticky, card--bordered etc.,)
    - Modifiers are generally separated by -- from block/element.

BEM - cont.,

- BEM makes debugging a cakewalk.
- Every styleblock is global scoped so, we can reuse any styling anywhere.
- BEM strictly discourages using hierarchical CSS which breaks its very own rule of reusability
- Pros:
  - Every style blocks is reusable.
  - Easy to debug and semantically meaningful.
  - Code becomes flexible over time.
  - Easy and fun to write CSS
  - SCSS makes writing BEM absolutely easy and fun.
  - Style base turns very readable and maintainable and scalable.
- Cons
  - Subjective opinion that BEM makes classnames very verbose and unnecessarily long.
  - Takes time to Adapt to the mental model of thinking everything in terms of blocks, elements

Bem: example

```
.btn{
        Padding: 8px 18px;
        Color: #fff;
}
.btn__icon {
        font-size: 80%;
        Position: absolute;
        letft:4px;
        top: 4px;
}
.btn--primary {color: red;}
```

Block

A button

Element

Icon inside the button
So, __

modifier

A different version on button , so --

SMACSS- Scalable & Modular Architecture for CSS

- Everything we learnt so far dealt with How, but SMACSS deals with where along with How,Its mostly a modularity achievement through file organization
- SMACSS suggests all css to be categorized into five categories
  - Base - default styles like html, body, a, a:visited, font settings etc.,
  - Layout - Grid System, alignment system, header, footer, article etc., some devs use l-prefixing to specify layout classes.
  - Modules - reusable module/ component/ objects etc., This is majority your CSS
  - States - States are variations possible for any element in css, like :active, :hover etc., or media queries, pseudo classes etc., SMACSS convention is to use is/has as prefixes for states., ex: is-active, is-loading, is-hidden, is-pressed etc.,
  - Theme - Any theme specific CSS if your project have. Large projects generally have same components look different based on pages, sections etc., those can be customized here
- Child Items in SMACSS generally have parent classname as prefix like .card, .card-title etc.,
- SMACSS leaves the naming convention to Developers, you can even use BEM convention with SMACSS.

SMACSS- Scalable & Modular Architecture for CSS

- Pros:
  - Less Code Repetition
  - consistent experience
  - Easier maintenance
  - Can be clubbed with BEM, OOCSS etc in modules
- Cons:
  - It is sometimes difficult to decide which category some styles fall into.,
  - For example, when using BEM, which category does :hover written in any module belong to, it can belong to where it can be easily written or separately in states
  - Some devs feels its unnecessarily bloating the classnames in html, and some time very long classnames
    - Ex: card--isactive__action--like etc.,

ITCSS - Inverted Triangle CSS

- ITCSS stands for *Inverted Triangle CSS* and it helps you to organize your project CSS files in such a way that you can better deal with (not always easy-to-deal with) CSS specifics like global namespace, cascade and selectors specificity
- ITCSS can be used with preprocessors or without them and is compatible with CSS methodologies like BEM, SMACSS or OOCSS.
- ITCSS recommends to separate your css into different Layers/sections
- Not a framework or library to download and use
- To fix problems because of CSS, problems because of Developers
- Strictly discourages using ids for styling

ITCSS - Inverted Triangle CSS

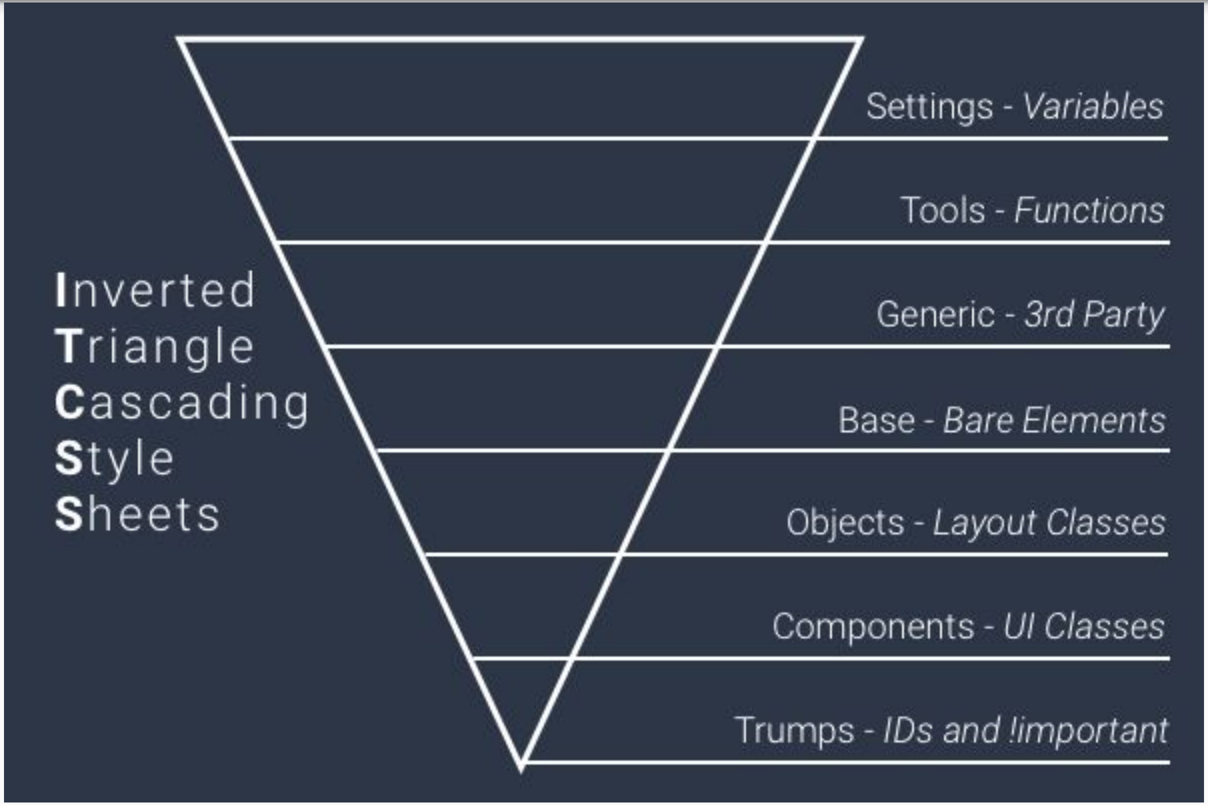

Image credit : Mikey Murphy ‘s medium article

ITCSS - Inverted Triangle CSS

- Settings:
    - Used with preprocessors Site wide settings/ configs/ variables etc.,
- Tools:
    - Site wide tools like mixins, functions, scss libraries etc.,
- Generic
    - Low specificity styles like css resets, css normalisers etc., This is where actual css starts generating. Above this its all preprocessor config and tools
- Elements
    - Any element style declarations/overrides etc., like headings, anchors, lists etc.,

ITCSS - Inverted Triangle CSS

- Objects:
  - Class based Style Objects like navigations, avatars, comment boxes etc., we can use OOCSS objects, BEM modules etc., here
- Components:
  - Specific UI components, Buttons, Tooltips, Form elements etc.,Often UI Components are composed of Objects and ui components.
- Utilities
  - Helper/utility classes which have ability to override anything declared above., like is-hidden, is-active etc classes., overrides, spacing helpers, responsive styles etc.,

ITCSS - Inverted Triangle CSS

- Pros:
  - Style of an element is always defined from lowest specific styles to very specific styles, less prone to ui bugs.
  - Maintainability
  - Readability
  - Scalability
- Cons:
  - No open source documentation
  - Relatively smaller and newer community
  - Open to subjective interpretation

Hands On CSS

1. 12 grid size Responsive Grid with SCSS - Bootstrap like
   I.e., col-lg/md/sm/xs-1/12, rows, medea queries and more
2. Better Variable(Colors etc.,) management for when you have preprocessors

Repo Link : https://github.com/bhavan777/scss-workshop