FLUXTALK

WRITING SCALABLE CSS

Thomas Winter
thomas.winter@fluxfederation.com
21st of June 2019

flux

BOLDLY GO.

CONTENTS

SELECTORS

SELECTORS TL;DR

👍🏽 Use **single-level-deep classes** `.text-input {}`.

Avoid **nesting** when possible `.widget h3 {}, header ul li {}`.

Avoid unnecessary **qualifying** `ul.nav {}`.

Avoid **attribute selectors** `[role="navigation"] {}`.

Avoid **complexity** `.nav .nav-item {}`.

Avoid **IDs** `#main-nav`.

SELECTOR INTENT

Be clear with your reason for selecting something.

```
/* Bad */   header > ul {}    /* Better */   .nav-primary {}
```

Don’t rely on circumstance or coincidence.

```
/* Bad */   .promo a {}       /* Better */   .btn--promo {}
```

Don’t cast a wide net if **only trying to catch one thing**.

```
/* Bad */   aside {}          /* Better */   .aside {}
```

SELECTOR INTENT

```
/* It might feel that something like this is less work. */

.header nav > ul > li > a {}

/* But when one of your links needs to behave differently, you
will end up adding more specific styles to override the default
styles for this one difference. */


/* This is better: */

.nav-primary {}

.nav-primary__link {}
```

NAMING

Do not use classes to **describe content**.

Content describes itself.

Pick vaguely; **be more abstract**.

Always **aim for reuse**.

Use a **drill-down naming approach** from biggest to smallest.

NAMING

```
/* Can only be used in one place now. */
.btn--login {}

/* Lots of options for reuse! */
.btn--positive {}
```

DRILLDOWN NAMING

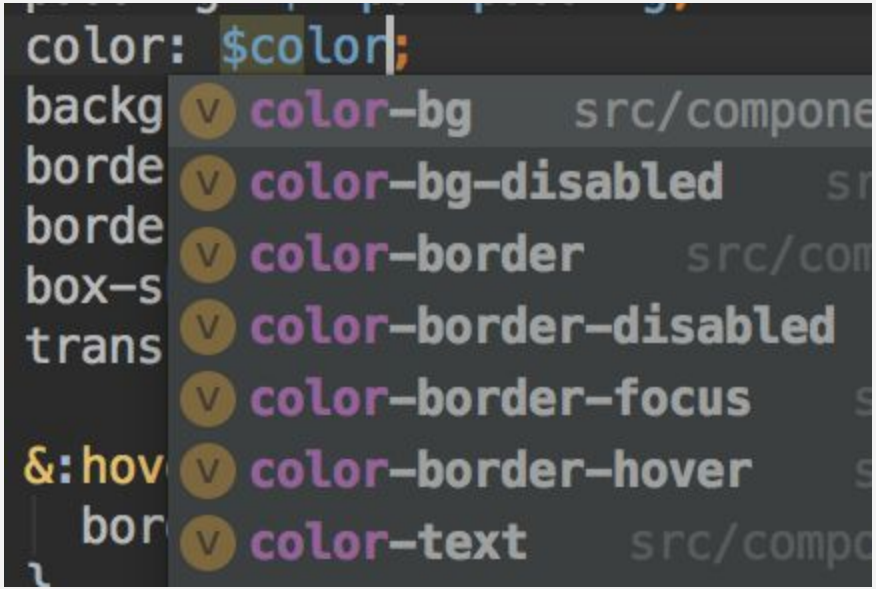
```
/* Bad. */
$base-color: #C0FFEE;
$primary-color: #BADA55;

.primary-nav {}
.secondary-nav {}
.inline-list {}
.positive-btn {}
```

```
/* Better! */
$color-base: #C0FFEE;
$color-primary: #BADA55;

.nav-primary {}
.nav-secondary {}
.list-inline {}
.btn-positive {}
```

DRILLDOWN NAMING



NAMESPACES

Help to write more **self-documenting** CSS.

Tell other developers **what classes are for**.

Increase **confidence**.

Tells us how classes **behave in a global sense**.

NAMESPACES

```
.apl- /* -> Styles live in the APL. */


.apl-component-name {}
.is-state {} /* .is-active, .is-readonly */
._hack {}
.qa-hook {} /* Even better: use data attributes */
.js-hook {} /* Even better: use data attributes */
```

NAMESPACES

```
.l-layout-object-name {}

.c-component-name {}

.u-utility-name {}

.t-theme-name {}
```

SPECIFICITY

SPECIFICITY

Specificity **determines, which CSS rule is applied** by the browsers.

If two selectors apply to the same element, the one with **higher specificity wins**.

When selectors have an **equal specificity** value, the **latest rule is the one that counts**.

CALCULATING SPECIFICITY

There are four distinct categories which define the specificity level of a given selector:

Element selector

Specificity: **0 0 1**

`a`, `p`, `li` ...

Class selector

Attribute selector

Specificity: **0 1 0**

`.text-large`

ID selector

Specificity: **1 0 0**

`#section-id`

Style attribute

Specificity: **1 0 0 0**

`<div style="...">`

CALCULATING SPECIFICITY – EXAMPLES

```
*                   /* 0 0 0 -> specificity =   0 */
LI                  /* 0 0 1 -> specificity =   1 */
UL OL+LI            /* 0 0 3 -> specificity =   3 */
UL OL LI.red        /* 0 1 3 -> specificity =  13 */
LI.red.level        /* 0 2 1 -> specificity =  21 */
#x34y               /* 1 0 0 -> specificity = 100 */
#x34y:not(FOO)      /* 1 0 1 -> specificity = 101 */
```

https://specificity.keegan.st/

CALCULATING SPECIFICITY

`!important` trumps it all.

The one and **only time** to use `!important` is for single purpose utility classes.

```
.font-weight-light {
    font-weight: 300 !important;
}
```

SPECIFICITY

Specificity is the **main cause of headaches** in CSS.

Keep it low at all times.

Avoid ID selectors.

Make heavy **use of classes**.

Aim for a **single-depth**-class-based architecture.

SPECIFICITY

```scss
/* This SCSS: */
.widget {
  .widget__title {
    .widget__date {}
  }
}
```

```scss
/* Will compile to: */
.widget {}
.widget .widget__title {}
.widget .widget__title .widget__date {}
```

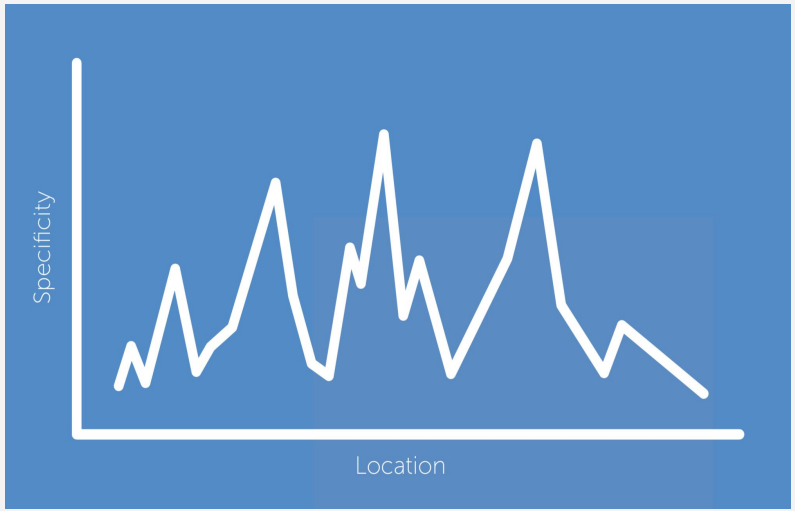
SPECIFICITY

```
/* Much better :) */
.widget {}
  .widget__title {}
  .widget__date {}
```

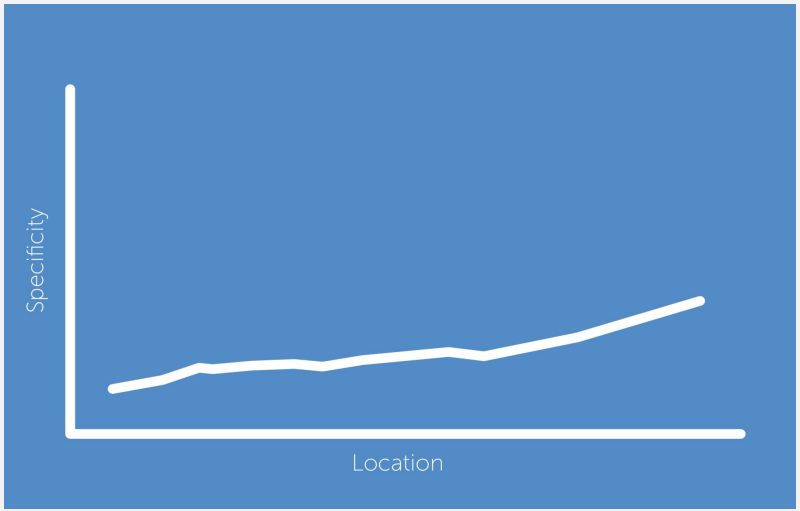
SPECIFICITY

SPECIFICITY GRAPH

Bad



That's better



https://jonassebastianohlsson.com/specificity-graph/

RE-USABILITY

SPOT REPETITION

Modularisation happens at **design-stage**.

Repetition means **consistency**.

Visual repetition is recyclable. That means **less code**...

...to **maintain**;

...to **go wrong**.

Building **bespoke things costs** more time and money.

But, **not everything** is or can be a component.

AVOID DEVIATION

“Can we just make this button a little bigger?”

– Always someone

“What’s the rule? Why is this one bigger? Will others need to be bigger? If so, do we make them all bigger? How do we handle different sized buttons in the future? Can we have a general button-but-bigger rule?”

– You

AVOID DEVIATION

```
/* Bad. */
.header .btn {
    padding: 2em;
}
```

```
/* Good! */
.btn {
    ...
}
.btn--large {
    padding: 2em;
}

/* or, if you have to */
```

CONTENT VS CONTEXT

Break things down into building blocks (**context**).

Then fill these with other things (**content**).

Let's look at the **card** example...

CONTENT VS CONTEXT

```
<!-- Bad -->
<div class="card">
    <div class="card__figure">...</div>
    <h4 class="card__title">...</h4>
    <a href="#" class="card__btn">...</a>
</div>
```

CONTENT VS CONTEXT

```
<!-- Better -->
<div class="card">
    <div class="card__figure">...</div>
    <div class="card__body">
        <h4>...</h4>
        <a href="#" class="btn">...</a>
    </div>
</div>
```

METHODOLOGIES

BEM (Block, Element, Modifier)

“BEM — Block Element Modifier is a methodology that helps you to create reusable components and code sharing in front-end development”

– http://getbem.com/

BEM NAMING

```
/* Block */
.person {}

/* Element of that block */
.person__leg {}

/* A modifier */
.person--tall {}
```

BEM NAMING

```
/* Style an Element based on a modifier: */
.person__leg {
    ...
    .person--tall & {
        ...
    }
}
```

BEM NAMING

```
/* Modify an Element directly: */

.person__leg--left {

    ...

}
```

BEM NAMING

```
/* Don't go all the way down the DOM tree: */
.person__arm__hand__finger {
    ...
}


/* Consider splitting a component up into smaller pieces */
```

BEM NAMING

```
/* Card example again */
<div class="card">
    <div class="card__figure">...</div>
    <div class="card__body">
        <h4>...</h4>
        <a href="#" class="btn btn--large">...</a>
    </div>
</div>
```

ITCSS – INVERTED TRIANGLE

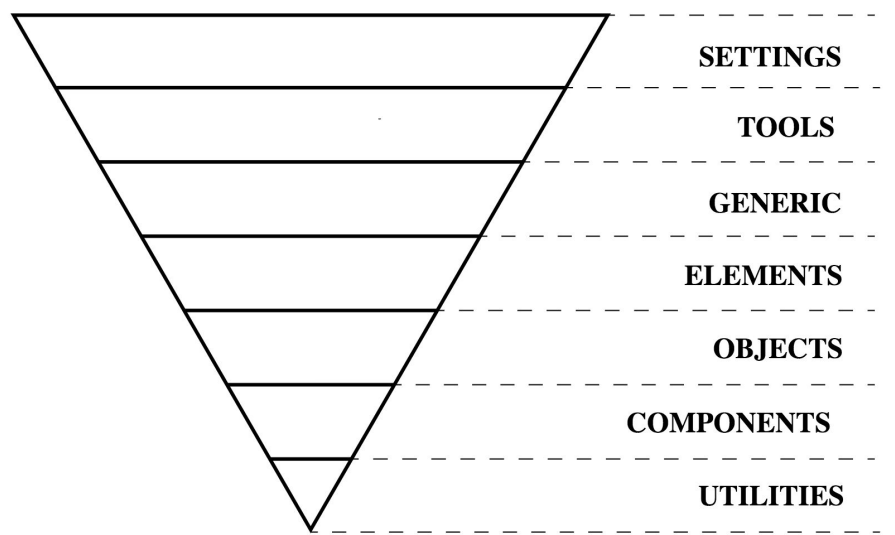
A sane, scalable, managed **architecture** for large UI projects.

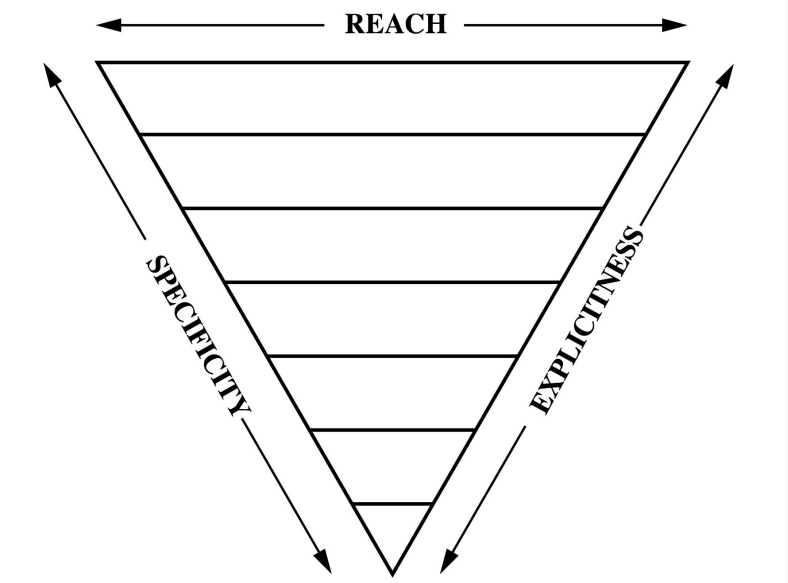
Manage the **dependency tree**.

Set the source order.

Tame inheritance and the cascade.

ITCSS – INVERTED TRIANGLE





UNITS

ABSOLUTE UNITS

`px` is the main absolute and **easiest to use** unit.

Try to limit the use of fixed size units, as they **don't scale**.

Most of the units you will be using will be **rendered in pixels**.

Use pixels for **border widths** and box-shadows.

RELATIVE UNITS – REM

`rem` is **relative to the root element**'s (html) font-size.

Provides **accessible scalability** and is easy to predict.

If font-size of the root element is 16px, then 1rem equals 16px.

For **font-sizes** the use of rems is generally preferred.

Margins and padding for layout objects such as gutters and sections should be defined in rem.

Example: https://codepen.io/mr__winter/pen/gNrWrx

RELATIVE UNITS – EM

`em` is **relative to the current font-size** of the element.

They are **difficult to predict** in nested scenarios (font sizes are inherited from parent elements).

A valuable measurement as they allow us to **scale elements in relation** to their parents.

A common use case for ems are **margins and padding for text based components** such as buttons and pills.

RELATIVE UNITS – OTHER USEFUL UNITS

`%` – Relative to the parent element.

`vw` – Relative to 1% of the width of the viewport*.

`vh` – Relative to 1% of the height of the viewport*.

`ex` – Relative to the x-height of the current font.

`ch` – Relative to the width of the “0” (zero).

* The viewport is the area of the browser window in which content can be seen.

UNITLESS VALUES

To completely **remove** the border, margin or padding from an element, you can use a unitless `0`.

Use unitless values with **line-heights**, like: `line-height: 1.25`.

SASS

BASICS

Split your code into partials, starting with underscores `_alerts.scss`.

Avoid using `@extend`. They can produce unexpected CSS output.

`@mixin`s are great for **generating repetitive styles** (buttons, theming, ...)

`@function`s make it easy to **abstract out common formulas**.

Variables are global, unless they are declared in a style block `{ … }`.

`!default` assigns a value to a variable, if that variable isn't defined yet.

See https://sass-lang.com/documentation for docs.

@EXTEND CHANGING SELECTOR ORDER

```
.one {
    color: red;
}
.two {
    color: green;
}
.three {
    @extend .one;
}
```

```
/* compiles to */
.one, .three {
  color: red;
}
.two {
  color: green;
}
```

```
<div class="three two">green, not red</div>
```

@EXTEND CAN CREATE OVERLY LONG OUTPUT

```scss
/* _message.scss */
.message + .message {
  margin-top: .5em;
}

.message-error {
  @extend .message;
}
```

```scss
/* compiles to */
.message + .message,
.message-error + .message-error,
.message + .message-error,
.message-error + .message {
  margin-top: .5em;
}
```

!DEFAULT VARIABLES

```scss
/* _library.scss */
$color: #000 !default;
.some-class {
  color: $color;
}
/* style.scss */
$black: #222;
@import 'library';
```

```scss
/* compiles to style.css */
.some-class {
  color: #222;
}
```

SASS VARIABLES VS CSS VARIABLES

Both are great, each for different use cases. Know the differences.

Sass variables are **imperative**.

CSS variables are **declarative**.

Sass variables are **compiled away** into CSS.

CSS variables can be **defined outside** the stylesheet where they are being used (<style> tags, other files, ...)

PRINCIPLES

MAKE EVERYTHING OPT-IN

```
.btn {
    padding: 1em;
}
```

```
/* This is dictatorial :( */
.sidebar .btn {
    padding: 2em;
}
```

```
/* This is opt-in :) */
.btn--large {
    padding: 2em;
}
```

IMMUTABILITY

Certain rules should **not be able to mutate**.

Utility classes are a perfect example (a helper class to hide something should always hide something).

The one and **only time** to use `!important`!

IMMUTABILITY

```
.profile--large .profile__img {
    display: block;
}
.display-none {
    display: none;
}
```

IMMUTABILITY

```
.profile--large .profile__img {
    display: block;
}
.display-none {
    display: none !important;
}
```

Q&A

FLUXY MAY BE ABLE TO HELP...