CSS Architecture

BBC—March, 2014

Agenda

A day of two halves

Morning

What's wrong with CSS?

OOCSS and BEM – a recap.

Architecting from scratch.

CSS architectures.

Writing and reading code.

Managing layout.

Afternoon

Over to you!

Applying different rules in different contexts.

Theming of components/modules.

Controlling layout of modules.

Advanced BEM.

Harry Roberts

Consultant Front-end Architect


sky

Warm up

Just to get us going

The problems with CSS

And, let’s be honest, there are a lot...

The problems with CSS

It’s old. Really old.

It can’t be changed.

It operates in a global namespace.

It’s based on inheritance.

It’s very, very loose.

Source order is critical.

The cascade needs managing.

Specificity negates all of the above.

It’s old. Really old.

CSS was written in a completely different era.

The web, and websites, have got a lot more complex.

We’re building 2014 products with ’90s tools.

It can’t be changed

We can add as much new stuff as we like...

...but we can never change it.

We can’t control the environment in which it runs.

It operates in a global namespace

CSS is leakier than the Titanic.

JS has things like IIFEs to execute code in a scope/sandbox.

There is no real way to fully encapsulate CSS.

It’s based on inheritance

Things get passed on to their descendants (the cascade).

We can’t control what does and doesn’t get passed on.

It’s very, very loose

It was designed to be easy to learn and use: yay!

Anyone/everyone can use it: d’oh!

‘Yeah, I can write CSS.’

‘No, you can *type* CSS.’

Most people know enough to be dangerous.

Writing CSS well is down to self-discipline, not the language.

Source order is critical

CSS is essentially procedural.

It needs to be written in a certain order.

Plonking rules on to the end of a stylesheet is a BadIdea™.

The cascade needs managing

Some things are passed down to descendants…

…some things aren’t…

Some things we want to be passed down…

…and some we don’t.

The cascade needs a lot of management and attention.

Specificity negates all of the above

Source order matters, and the cascade needs managing.
Until specificity.

Weighty selectors can undo all of the effects of the cascade, source order, and inheritance.

It’s like CSS doesn’t even get on with itself.

Where does that leave us?

With an old, loose, leaky, globally-operating, inheritance-based language which is entirely dependent on source-order, except when you introduce its own worst feature: *specificity*.

Where does that leave us?

Not very suitable for building large front-ends with...

OOCSS

Structure and skin

OOCSS

Abstracted design patterns.

Structure recyclable without skin.

Very DRY.

Promotes consistency.

BEM

A new front-end methodology

BEM

More than just a naming convention...

...but the naming convention is its best feature.

Very transparent.

Very readable.

More useful in HTML than it is in CSS.

```
<div class="block  block--modifier">
   <span class="block__element">…</span>
</div>
```

```
<div class="box  profile  pro-user">
    <img class="avatar  image" />
    <p class="bio">...</p>
</div>
```

```
<div class="box  profile  profile--is-pro-user">
    <img class="avatar  profile__image" />
    <p class="profile__bio">...</p>
</div>
```

Architecting from scratch

Where to start

“We’ll get it right this time…”

Architecting from scratch

Always a nice position to be in.

Think about what went wrong before...

...and how you'll avoid it this time.

A lot of thinking time.

Preempt architectural/scalability problems.


NHS

NHS

Half a day to get the entire CSS architecture sorted.

One day to rip it all down and start again.

Solving architectural woes is very, very important.

NHS

Many things are constant from project to project.

Solid architecture is relatively project agnostic.

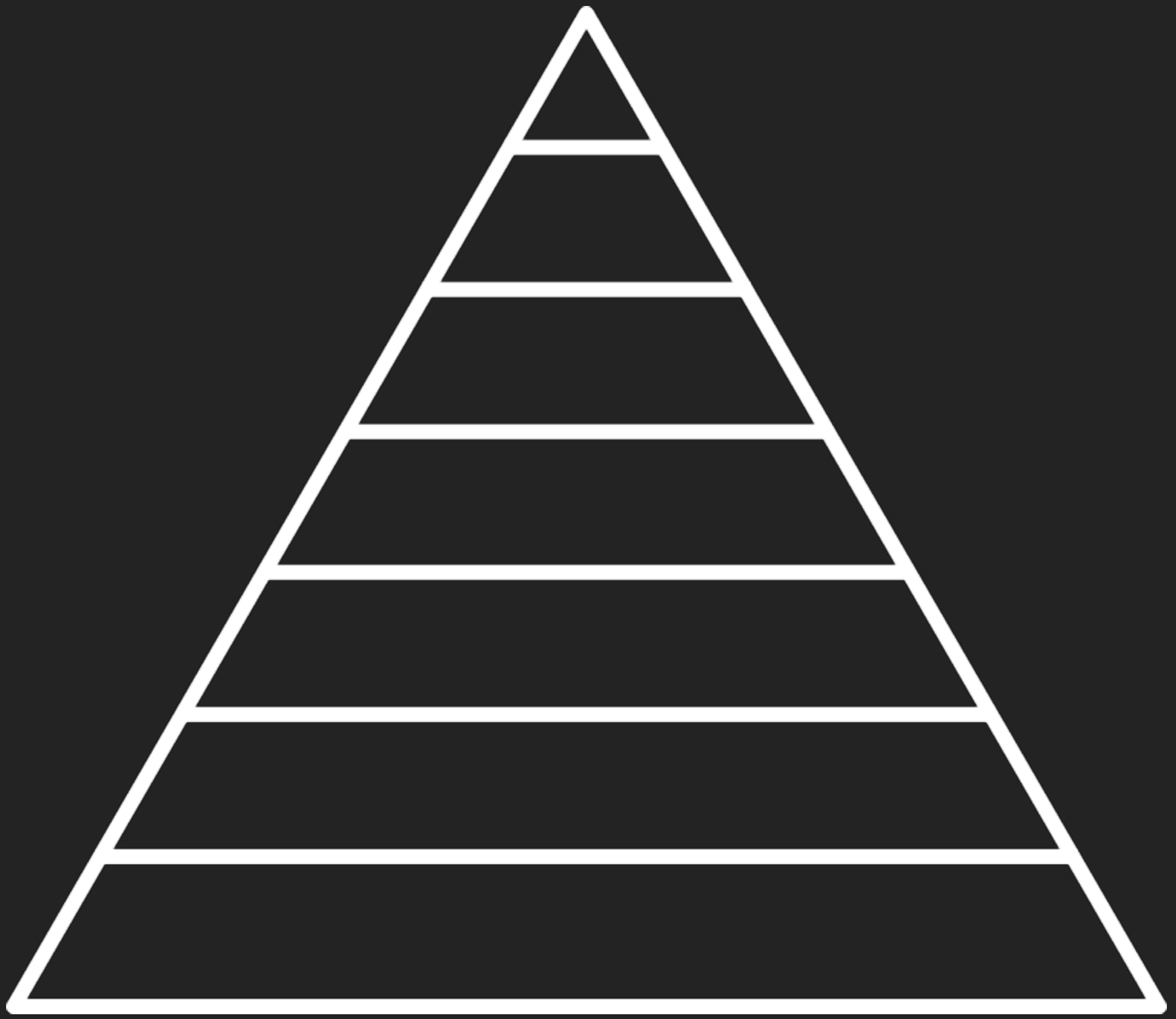
If you have a particularly unusual set of requirements, pick and choose the bits that still apply.

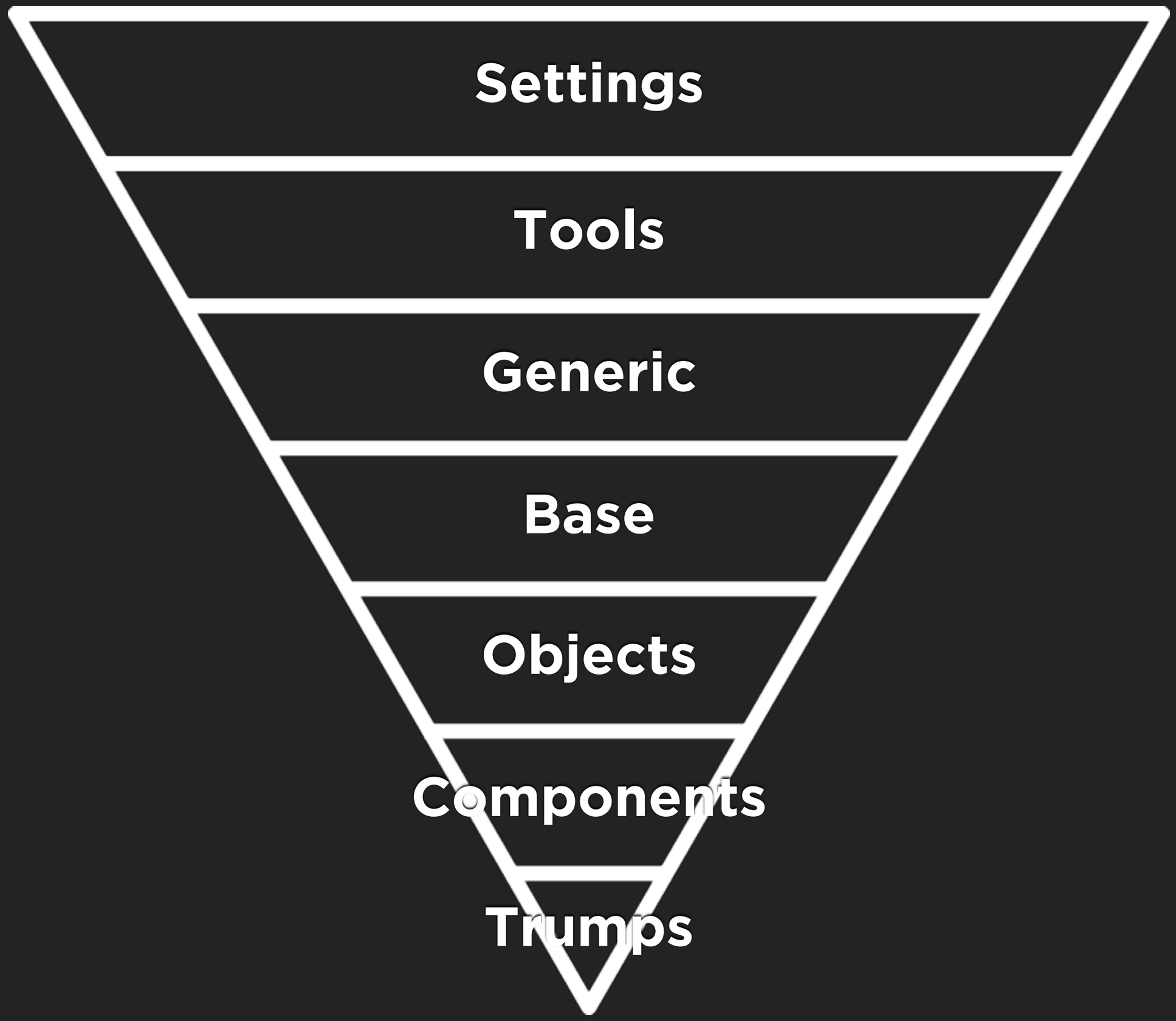
CSS Architectures

Ways of writing, organising, and scaling CSS

The Inverted Triangle

A sane, managed architecture for CSS




Settings
Tools
Generic
Base
Objects
Components
Trumps

Inverted Triangle

Settings – Sass/LESS config, globals, etc.

Tools – Mixins, functions, etc.

Generic – Reset, Normalize.css, global box-sizing, etc.

Base – Unclassed HTML elements, type selectors.

Objects – Objects and abstractions.

Components – Self-contained, designed modules.

Trumps – Overrides and helpers.

Inverted Triangle

Managed architecture for the entire project.

Sane source order.

Tamed inheritance.

Efficient cascade.

Controlled specificity.

Very terse.

Scalable in many directions.

Inverted Triangle

Components themselves are not very portable.

Although managed, it’s built on dependency.

Reliant on the cascade.

Decoupled architecture

A self-contained non-architecture for CSS

Decoupled architecture

No dependencies.

Minimal inheritance.

Self sufficient.

Extremely portable.

Decoupled architecture

Very verbose.

Needs to be defensive.

Lots of repetition at project level.

Doesn’t work very well with OOCSS.

Writing and reading code

Tips and tricks for writing HTML & CSS for others...

Write CSS in specificity order

Don’t group rules into roles, write them as they’re applied

Write CSS in specificity order

Write the lowest-specificity, most far-reaching rules first.

Increase specificity and explicitness as you go down.

Each step down should cascade less and less.

The end should be the most explicit, specific rulesets.

Always focus on the little bits

Try avoid thinking about the bigger picture

“Take care of the pennies, and the pounds will take care of themselves.”

“Take care of the bits, and the bytes will take care of themselves.”

@fold_left

Always focus on the little bits

Sounds a little contradictory, but...

...once you've solved your architectural stuff, stop thinking about the bigger picture.

Keep code ignorant, self-contained, self-sufficient.

Little and often.

A brickmaker never concerns himself with how big a building might be...

...and a builder never complains that she needs too many bricks.

Always focus on the little bits

Make all files/components/objects/partials/everything as small as possible, if not smaller.

Package any settings up into the partial itself.

Make everything highly configurable without having to touch it.

Pepper bits of code into your project, rather than dropping a black-box into the middle of it.

inuitcss

main.scss

```
@include "settings.global";
@include "tools.mixins";
@include "generic.reset";
@include "base.headings";

$media-gutter:  10px;
@include "objects.media";

$mini-bio-color-background:  #BADA55;
@include "components.mini-bio";
```

_objects.media.scss

```
$media-gutter:  18px !default;

.media {
    ...
}
```

Ignorance is bliss

Make sure code doesn’t know about anything else

Ignorance is bliss

Keeping code ignorant and dumb inherently makes it more self sufficient.

Any dependencies you do have need to be carefully considered.

Everything is subject to change

Nothing is forever, things can and will change, and that’s okay.

Everything is subject to change

You might not get it right the first time, but it's important to revisit and refactor.

Nothing_is_final_v2_EDITED_new.psd

Write code with this in mind.

Be as accommodating as you can be.

'Now I have to maintain classes in my markup…'

Grouping classes

Bringing order to class-heavy markup

```
<div class="[ media  media--large ]  album-info">
```

Meaningful names

Add meaning to non-descriptive classes

```
<ul class="ui-list">
    <li class="ui-list__item">
        <div class="media">
            ...
        </div>
    </li>
</ul>
```

```
<ul class="ui-list" data-ui-component="friends-list">
    <li class="ui-list__item">
        <div class="media">
            ...
        </div>
    </li>
</ul>
```

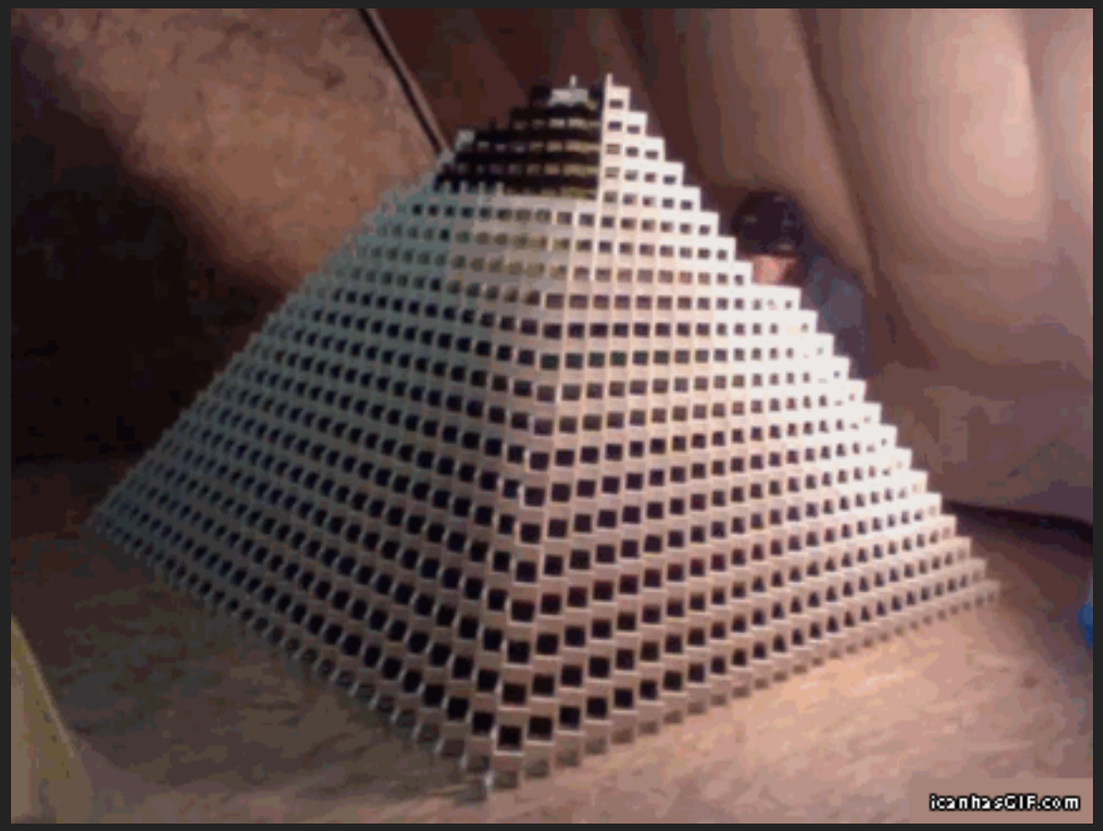
```
.show-components [data-ui-component] {
    outline: 5px solid yellow;
}
```

Managing layout

Extra DIVs and insemantic classes!

“There’s no point building a fluid, decoupled suite of UI components it you’re going to apply a load of widths and floats to them.”

Your UI without a layout system



Your UI with a layout system

Managing layout

Layout systems (not grid systems).

Nudging layout on a micro-level.

Managing layout

Allows you to control your UI's layout from one location.

Rapidly arrange UIs using components as building blocks.

Components are your bricks, a layout system is your mortar.

Layout code becomes easier to find (and debug).

Componentise your layout.

```
<div class="[ grid  grid--wide ]">
  <div class="[ grid__item  one-quarter ]">
    <article class="media  mb" data-ui-component="mini-bio">
      ...
    </article>
  </div>
</div>
```

```
<div class="[ grid  grid--wide ]">
  <div class="[ grid__item  one-quarter ]">
    <article class="media  mb" data-ui-component="mini-bio">
      ...
    </article>
  </div>
</div>
```

```
<div class="[ grid  grid--wide ]">
  <div class="[ grid__item  one-quarter ]">
    <article class="media  mb" data-ui-component="mini-bio">
      ...
    </article>
  </div>
</div>
```

```
<div class="[ grid  grid--wide ]">
  <div class="[ grid__item  one-quarter ]">
    <article class="media  mb" data-ui-component="mini-bio">
      ...
    </article>
  </div>
</div>
```

```
<div class="[ grid  grid--wide ]">
  <div class="[ grid__item  one-quarter ]">
    <article class="media  mb" data-ui-component="mini-bio">
      ...
    </article>
  </div>
</div>
```

Fin!

Questions?