css { guide: lines; }

High-level advice and guidelines for writing sane, manageable, scalable CSS

About the Author

CSS Guidelines is a document by me, **Harry Roberts**. I am a Consultant Front-

end Architect from the UK, and I help companies all over the world write and manage better quality UIs for their products and teams. I am available for hire.

Follow me on Twitter or **Hire Me**

Support the Guidelines

CSS Guidelines is provided through a pay-what-you-like model—from $0 upward. If *CSS Guidelines* is useful to you or your team, please consider **supporting it**.

Support the Guidelines

Get updates about changes, additions, and new and upcoming sections by following **@cssguidelines** on Twitter.

Contents

- Introduction
  - The Importance of a Styleguide
  - Disclaimers
- Syntax and Formatting
  - Multiple Files
  - Table of Contents

        - 80 Characters Wide
        - Titling
        - Anatomy of a Ruleset
        - Multi-line CSS
        - Indenting
            - Indenting Sass
            - Alignment
        - Meaningful Whitespace
        - HTML
- Commenting
    - High-level
        - Object–Extension Pointers
    - Low-level
    - Preprocessor Comments
    - Removing Comments
- Naming Conventions
    - Hyphen Delimited
    - BEM-like Naming
        - Starting Context
        - More Layers
        - Modifying Elements
    - Naming Conventions in HTML
    - JavaScript Hooks
        - `data-*` Attributes
    - Taking It Further
- CSS Selectors
    - Selector Intent
    - Reusability
    - Location Independence
    - Portability
        - Quasi-Qualified Selectors
    - Naming
        - Naming UI Components
    - Selector Performance

        - The Key Selector
    - General Rules
- Specificity
    - IDs in CSS
    - Nesting
    - `!important`
    - Hacking Specificity
- Architectural Principles
    - High-level Overview
    - Object-orientation
    - The Single Responsibility Principle
    - The Open/Closed Principle
    - DRY
    - Composition over Inheritance
    - The Separation of Concerns
        - Misconceptions

Up Next

- Preprocessors
- Layout
- Performance
- Sanity, Simplicity
- Code Smells
- Legacy, Hacks, and Technical Debt

Introduction

CSS is not a pretty language. While it is simple to learn and get started with, it soon becomes problematic at any reasonable scale. There isn't much we can do

to change how CSS works, but we can make changes to the way we author and structure it.

In working on large, long-running projects, with dozens of developers of differing specialities and abilities, it is important that we all work in a unified way in order to—among other things—

- keep stylesheets maintainable;
- keep code transparent, sane, and readable;
- keep stylesheets scalable.

There are a variety of techniques we must employ in order to satisfy these goals, and *CSS Guidelines* is a document of recommendations and approaches that will help us to do so.

The Importance of a Styleguide

A coding styleguide (note, not a visual styleguide) is a valuable tool for teams who

- build and maintain products for a reasonable length of time;
- have developers of differing abilities and specialisms;
- have a number of different developers working on a product at any given time;
- on-board new staff regularly;
- have a number of codebases that developers dip in and out of.

Whilst styleguides are typically more suited to product teams—large codebases on long-lived and evolving projects, with multiple developers contributing over prolonged periods of time—all developers should strive for a degree of standardisation in their code.

A good styleguide, when well followed, will

- set the standard for code quality across a codebase;
- promote consistency across codebases;
- give developers a feeling of familiarity across codebases;
- increase productivity.

Styleguides should be learned, understood, and implemented at all times on a project which is governed by one, and any deviation must be fully justified.

Disclaimers

CSS Guidelines is *a* styleguide; it is not *the* styleguide. It contains methodologies, techniques, and tips that I would firmly recommend to my clients and teams, but your own tastes and circumstances may well be different. Your mileage may vary.

These guidelines are opinionated, but they have been repeatedly tried, tested, stressed, refined, broken, reworked, and revisited over a number of years on projects of all sizes.

Syntax and Formatting

One of the simplest forms of a styleguide is a set of rules regarding syntax and formatting. Having a standard way of writing (*literally* writing) CSS means that code will always look and feel familiar to all members of the team.

Further, code that looks clean *feels* clean. It is a much nicer environment to work in, and prompts other team members to maintain the standard of cleanliness that they found. Ugly code sets a bad precedent.

At a very high-level, we want

- four (4) space indents, no tabs;
- 80 character wide columns;
- multi-line CSS;
- meaningful use of whitespace.

But, as with anything, the specifics are somewhat irrelevant—consistency is key.

Multiple Files

With the meteoric rise of preprocessors of late, more often is the case that developers are splitting CSS across multiple files.

Even if not using a preprocessor, it is a good idea to split discrete chunks of code into their own files, which are concatenated during a build step.

If, for whatever reason, you are not working across multiple files, the next sections might require some bending to fit your setup.

Table of Contents

A table of contents is a fairly substantial maintenance overhead, but the benefits it brings far outweigh any costs. It takes a diligent developer to keep a table of contents up to date, but it is well worth sticking with. An up-to-date table of contents provides a team with a single, canonical catalogue of what is in a CSS project, what it does, and in what order.

A simple table of contents will—in order, naturally—simply provide the name of the section and a brief summary of what it is and does, for example:

```
/**
 * CONTENTS
 *
```

```
 * SETTINGS
 * Global..............Globally-available variables and config.
 *
 * TOOLS
 * Mixins..............Useful mixins.
 *
 * GENERIC
 * Normalize.css........A level playing field.
 * Box-sizing...........Better default `box-sizing`.
 *
 * BASE
 * Headings.............H1–H6 styles.
 *
 * OBJECTS
 * Wrappers.............Wrapping and constraining elements.
 *
 * COMPONENTS
 * Page-head............The main page header.
 * Page-foot............The main page footer.
 * Buttons..............Button elements.
 *
 * TRUMPS
 * Text.................Text helpers.
 */
```

Each item maps to a section and/or include.

Naturally, this section would be substantially larger on the majority of projects, but hopefully we can see how this section—in the master stylesheet—provides developers with a project-wide view of what is being used where, and why.

80 Characters Wide

Where possible, limit CSS files' width to 80 characters. Reasons for this include

- the ability to have multiple files open side by side;
- viewing CSS on sites like GitHub, or in terminal windows;
- providing a comfortable line length for comments.

```
/**
 * I am a long-form comment. I describe, in detail, the CSS that fo
 * such a long comment that I easily break the 80 character limit,
 * broken across several lines.
 */
```

There will be unavoidable exceptions to this rule—such as URLs, or gradient syntax—which shouldn't be worried about.

Titling

Begin every new major section of a CSS project with a title:

```
/*------------------------------------*\
    #SECTION-TITLE
\*------------------------------------*/

.selector {}
```

The title of the section is prefixed with a hash (`#`) symbol to allow us to perform more targeted searches (e.g. `grep`, etc.): instead of searching for just `SECTION-TITLE`—which may yield many results—a more scoped search of `#SECTION-TITLE` should return only the section in question.

Leave a carriage return between this title and the next line of code (be that a comment, some Sass, or some CSS).

If you are working on a project where each section is its own file, this title should appear at the top of each one. If you are working on a project with multiple sections per file, each title should be preceded by five (5) carriage returns. This extra whitespace coupled with a title makes new sections much easier to spot when scrolling through large files:

```
/*----------------------------------*\
    #A-SECTION
\*----------------------------------*/

.selector {}




/*----------------------------------*\
    #ANOTHER-SECTION
\*----------------------------------*/

/**
 * Comment
 */

.another-selector {}
```

Anatomy of a Ruleset

Before we discuss how we write out our rulesets, let's first familiarise ourselves with the relevant terminology:

```
[selector] {
```

```
    [property]: [value];
    [<--declaration--->]
}
```

For example:

```
.foo, .foo--bar,
.baz {
    display: block;
    background-color: green;
    color: red;
}
```

Here you can see we have

- related selectors on the same line; unrelated selectors on new lines;
- a space before our opening brace (`{`);
- properties and values on the same line;
- a space after our property–value delimiting colon (`:`);
- each declaration on its own new line;
- the opening brace (`{`) on the same line as our last selector;
- our first declaration on a new line after our opening brace (`{`);
- our closing brace (`}`) on its own new line;
- each declaration indented by four (4) spaces;
- a trailing semi-colon (`;`) on our last declaration.

This format seems to be the largely universal standard (except for variations in number of spaces, with a lot of developers preferring two (2)).

As such, the following would be incorrect:

```
.foo, .foo--bar, .baz
```

```
{
        display:block;
        background-color:green;
        color:red }
```

Problems here include

- tabs instead of spaces;
- unrelated selectors on the same line;
- the opening brace (`{`) on its own line;
- the closing brace (`}`) does not sit on its own line;
- the trailing (and, admittedly, optional) semi-colon (`;`) is missing;
- no spaces after colons (`:`).

Multi-line CSS

CSS should be written across multiple lines, except in very specific circumstances. There are a number of benefits to this:

- A reduced chance of merge conflicts, because each piece of functionality exists on its own line.
- More 'truthful' and reliable `diff`s, because one line only ever carries one change.

Exceptions to this rule should be fairly apparent, such as similar rulesets that only carry one declaration each, for example:

```
.icon {
    display: inline-block;
    width:  16px;
    height: 16px;
    background-image: url(/img/sprite.svg);
}
```

```
.icon--home     { background-position:   0     0  ; }
.icon--person   { background-position: -16px   0  ; }
.icon--files    { background-position:   0   -16px; }
.icon--settings { background-position: -16px -16px; }
```

These types of ruleset benefit from being single-lined because

- they still conform to the one-reason-to-change-per-line rule;
- they share enough similarities that they don't need to be read as thoroughly as other rulesets—there is more benefit in being able to scan their selectors, which are of more interest to us in these cases.

Indenting

As well as indenting individual declarations, indent entire related rulesets to signal their relation to one another, for example:

```
.foo {}

    .foo__bar {}

        .foo__baz {}
```

By doing this, a developer can see at a glance that `.foo__baz {}` lives inside `.foo__bar {}` lives inside `.foo {}`.

This quasi-replication of the DOM tells developers a lot about where classes are expected to be used without them having to refer to a snippet of HTML.

Indenting Sass

Sass provides nesting functionality. That is to say, by writing this:

```
.foo {
    color: red;

    .bar {
        color: blue;
    }

}
```

...we will be left with this compiled CSS:

```
.foo { color: red; }
.foo .bar { color: blue; }
```

When indenting Sass, we stick to the same four (4) spaces, and we also leave a blank line before and after the nested ruleset.

N.B. Nesting in Sass should be avoided wherever possible. See **the Specificity section** for more details.

Alignment

Attempt to align common and related identical strings in declarations, for example:

```
.foo {
    -webkit-border-radius: 3px;
       -moz-border-radius: 3px;
            border-radius: 3px;
```

```
}

.bar {
    position: absolute;
    top:    0;
    right:  0;
    bottom: 0;
    left:   0;
    margin-right: -10px;
    margin-left:  -10px;
    padding-right: 10px;
    padding-left:  10px;
}
```

This makes life a little easier for developers whose text editors support column editing, allowing them to change several identical and aligned lines in one go.

Meaningful Whitespace

As well as indentation, we can provide a lot of information through liberal and judicious use of whitespace between rulesets. We use:

- One (1) empty line between closely related rulesets.
- Two (2) empty lines between loosely related rulesets.
- Five (5) empty lines between entirely new sections.

For example:

```
/*------------------------------------*\
    #FOO
\*------------------------------------*/

.foo {}

    .foo__bar {}


.foo--baz {}




/*------------------------------------*\
    #BAR
\*------------------------------------*/

.bar {}

    .bar__baz {}

    .bar__foo {}
```

There should never be a scenario in which two rulesets do not have an empty line between them. This would be incorrect:

```
.foo {}
    .foo__bar {}
```

```
.foo--baz {}
```

HTML

Given HTML and CSS' inherently interconnected nature, it would be remiss of me to not cover some syntax and formatting guidelines for markup.

Always quote attributes, even if they would work without. This reduces the chance of accidents, and is a more familiar format to the majority of developers. For all this would work (and is valid):

```
<div class=box>
```

...this format is preferred:

```
<div class="box">
```

The quotes are not required here, but err on the safe side and include them.

When writing multiple values in a class attribute, separate them with two spaces, thus:

```
<div class="foo  bar">
```

When multiple classes are related to each other, consider grouping them in square brackets (`[` and `]`), like so:

```
<div class="[ box  box--highlight ]  [ bio  bio--long ]">
```

This is not a firm recommendation, and is something I am still trialling myself, but it does carry a number of benefits. Read more in ***Grouping related classes in your markup***.

As with our rulesets, it is possible to use meaningful whitespace in your HTML. You can denote thematic breaks in content with five (5) empty lines, for example:

```
<header class="page-head">
    ...
</header>




<main class="page-content">
    ...
</main>




<footer class="page-foot">
    ...
</footer>
```

Separate independent but loosely related snippets of markup with a single empty line, for example:

```
<ul class="primary-nav">
```

```
    <li class="primary-nav__item">
        <a href="/" class="primary-nav__link">Home</a>
    </li>

    <li class="primary-nav__item  primary-nav__trigger">
        <a href="/about" class="primary-nav__link">About</a>

        <ul class="primary-nav__sub-nav">
            <li><a href="/about/products">Products</a></li>
            <li><a href="/about/company">Company</a></li>
        </ul>

    </li>

    <li class="primary-nav__item">
        <a href="/contact" class="primary-nav__link">Contact</a>
    </li>

</ul>
```

This allows developers to spot separate parts of the DOM at a glance, and also allows certain text editors—like Vim, for example—to manipulate empty-line-delimited blocks of markup.

Further Reading

- ***Grouping related classes in your markup***

Commenting

The cognitive overhead of working with CSS is huge. With so much to be aware of, and so many project-specific nuances to remember, the worst situation most developers find themselves in is being the-person-who-didn't-write-this-code. Remembering your own classes, rules, objects, and helpers is manageable *to an extent*, but anyone inheriting CSS barely stands a chance.

CSS needs more comments.

As CSS is something of a declarative language that doesn't really leave much of a paper-trail, it is often hard to discern—from looking at the CSS alone—

- whether some CSS relies on other code elsewhere;
- what effect changing some code will have elsewhere;
- where else some CSS might be used;
- what styles something might inherit (intentionally or otherwise);
- what styles something might pass on (intentionally or otherwise);
- where the author intended a piece of CSS to be used.

This doesn't even take into account some of CSS' many quirks—such as various sates of `overflow` triggering block formatting context, or certain transform properties triggering hardware acceleration—that make it even more baffling to developers inheriting projects.

As a result of CSS not telling its own story very well, it is a language that really does benefit from being heavily commented.

As a rule, you should comment anything that isn't immediately obvious from the code alone. That is to say, there is no need to tell someone that `color: red;` will make something red, but if you're using `overflow: hidden;` to clear floats—as opposed to clipping an element's overflow—this is probably something worth documenting.

High-level

For large comments that document entire sections or components, we use a DocBlock-esque multi-line comment which adheres to our 80 column width.

Here is a real-life example from the CSS which styles the page header on **CSS Wizardry**:

```
/**
 * The site's main page-head can have two different states:
 *
 * 1) Regular page-head with no backgrounds or extra treatments; it
 *    contains the logo and nav.
 * 2) A masthead that has a fluid-height (becoming fixed after a ce
 *    which has a large background image, and some supporting text.
 *
 * The regular page-head is incredibly simple, but the masthead ver
 * slightly intermingled dependency with the wrapper that lives ins
 */
```

This level of detail should be the norm for all non-trivial code—descriptions of states, permutations, conditions, and treatments.

Object–Extension Pointers

When working across multiple partials, or in an OOCSS manner, you will often find that rulesets that can work in conjunction with each other are not always in the same file or location. For example, you may have a generic button object—which provides purely structural styles—which is to be extended in a component-level partial which will add cosmetics. We document this relationship across files with simple *object–extension pointers*. In the object file:

```
/**
 * Extend `.btn {}` in _components.buttons.scss.
 */

.btn {}
```

And in your theme file:

```
/**
 * These rules extend `.btn {}` in _objects.buttons.scss.
 */

.btn--positive {}

.btn--negative {}
```

This simple, low effort commenting can make a lot of difference to developers who are unaware of relationships across projects, or who are wanting to know how, why, and where other styles might be being inherited from.

Low-level

Oftentimes we want to comment on specific declarations (i.e. lines) in a ruleset. To do this we use a kind of reverse footnote. Here is a more complex comment detailing the larger site headers mentioned above:

```
/**
 * Large site headers act more like mastheads. They have a faux-flu
 * which is controlled by the wrapping element inside it.
 *
 * 1. Mastheads will typically have dark backgrounds, so we need to
```

```
 *    the contrast is okay. This value is subject to change as the
 *    image changes.
 * 2. We need to delegate a lot of the masthead's layout to its wra
 *    rather than the masthead itself: it is to this wrapper that m
 *    are positioned.
 * 3. The wrapper needs positioning context for us to lay our nav a
 *    text in.
 * 4. Faux-fluid-height technique: simply create the illusion of fl
 *    creating space via a percentage padding, and then position ev
 *    the top of that. This percentage gives us a 16:9 ratio.
 * 5. When the viewport is at 758px wide, our 16:9 ratio means that
 *    is currently rendered at 480px high. Let's…
 * 6. …seamlessly snip off the fluid feature at this height, and…
 * 7. …fix the height at 480px. This means that we should see no ju
 *    as the masthead moves from fluid to fixed. This actual value
 *    account the padding and the top border on the header itself.
 */

.page-head--masthead {
    margin-bottom: 0;
    background: url(/img/css/masthead.jpg) center center #2e2620;
    @include vendor(background-size, cover);
    color: $color-masthead; /* [1] */
    border-top-color: $color-masthead;
    border-bottom-width: 0;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1) inset;

    @include media-query(lap-and-up) {
        background-image: url(/img/css/masthead-medium.jpg);
    }

    @include media-query(desk) {
        background-image: url(/img/css/masthead-large.jpg);
    }
```

```
    > .wrapper { /* [2] */
        position: relative; /* [3] */
        padding-top: 56.25%; /* [4] */

        @media screen and (min-width: 758px) { /* [5] */
            padding-top: 0; /* [6] */
            height: $header-max-height - double($spacing-unit) - $h
        }

    }

}
```

These types of comment allow us to keep all of our documentation in one place whilst referring to the parts of the ruleset to which they belong.

Preprocessor Comments

With most—if not all—preprocessors, we have the option to write comments that will not get compiled out into our resulting CSS file. As a rule, use these comments to document code that would not get written out to that CSS file either. If you are documenting code which will get compiled, use comments that will compile also. For example, this is correct:

```
// Dimensions of the @2x image sprite:
$sprite-width:  920px;
$sprite-height: 212px;

/**
 * 1. Default icon size is 16px.
 * 2. Squash down the retina sprite to display at the correct size.
 */
```

```
.sprite {
    width:  16px; /* [1] */
    height: 16px; /* [1] */
    background-image: url(/img/sprites/main.png);
    background-size: ($sprite-width / 2 ) ($sprite-height / 2); /*
}
```

We have documented variables—code which will not get compiled into our CSS file—with preprocessor comments, whereas our CSS—code which will get compiled into our CSS file—is documented using CSS comments. This means that we have only the correct and relevant information available to us when debugging our compiled stylesheets.

Removing Comments

It should go without saying that no comments should make their way into production environments—all CSS should be minified, resulting in loss of comments, before being deployed.

Naming Conventions

Naming conventions in CSS are hugely useful in making your code more strict, more transparent, and more informative.

A good naming convention will tell you and your team

- what type of thing a class does;
- where a class can be used;
- what (else) a class might be related to.

The naming convention I follow is very simple: hyphen (-) delimited strings, with BEM-like naming for more complex pieces of code.

It's worth noting that a naming convention is not normally useful CSS-side of development; they really come into their own when viewed in HTML.

Hyphen Delimited

All strings in classes are delimited with a hyphen (-), like so:

```
.page-head {}

.sub-content {}
```

Camel case and underscores are not used for regular classes; the following are incorrect:

```
.pageHead {}

.sub_content {}
```

BEM-like Naming

For larger, more interrelated pieces of UI that require a number of classes, we use a BEM-like naming convention.

BEM, meaning *Block*, *Element*, *Modifier*, is a front-end methodology coined by developers working at Yandex. Whilst BEM is a complete methodology, here we are only concerned with its naming convention. Further, the naming convention here only is BEM-*like*; the principles are exactly the same, but the actual syntax differs slightly.

BEM splits components' classes into three groups:

- Block: The sole root of the component.
- Element: A component part of the Block.
- Modifier: A variant or extension of the Block.

To take an analogy (note, not an example):

```
.person {}
.person__head {}
.person--tall {}
```

Elements are delimited with two (2) underscores (`__`), and Modifiers are delimited by two (2) hyphens (`--`).

Here we can see that `.person {}` is the Block; it is the sole root of a discrete entity. `.person__head {}` is an Element; it is a smaller part of the `.person {}` Block. Finally, `.person--tall {}` is a Modifier; it is a specific variant of the `.person {}` Block.

Starting Context

Your Block context starts at the most logical, self-contained, discrete location. To continue with our person-based analogy, we'd not have a class like `.room__person {}`, as the room is another, much higher context. We'd probably have separate Blocks, like so:

```
.room {}

    .room__door {}

.room--kitchen {}
```

```
.person {}

    .person__head {}
```

If we did want to denote a `.person {}` inside a `.room {}`, it is more correct to use a selector like `.room .person {}` which bridges two Blocks than it is to increase the scope of existing Blocks and Elements.

A more realistic example of properly scoped blocks might look something like this, where each chunk of code represents its own Block:

```
.page {}


.content {}


.sub-content {}


.footer {}

    .footer__copyright {}
```

Incorrect notation for this would be:

```
.page {}

    .page__content {}
```

```
    .page__sub-content {}

    .page__footer {}

        .page__copyright {}
```

It is important to know when BEM scope starts and stops. As a rule, BEM applies to self-contained, discrete parts of the UI.

More Layers

If we were to add another Element—called, let's say, `.person__eye {}`—to this `.person {}` component, we would not need to step through every layer of the DOM. That is to say, the correct notation would be `.person__eye {}`, and not `.person__head__eye {}`. Your classes do not reflect the full paper-trail of the DOM.

Modifying Elements

You can have variants of Elements, and these can be denoted in a number of ways depending on how and why they are being modified. Carrying on with our person example, a blue eye might look like this:

```
.person__eye--blue {}
```

Here we can see we're directly modifying the eye Element.

Things can get more complex, however. Please excuse the crude analogy, and let's imagine we have a face Element that is handsome. The person themselves isn't that handsome, so we modify the face Element directly—a handsome face on a regular person:

```
.person__face--handsome {}
```

But what if that person *is* handsome, and we want to style their face because of that fact? A regular face on a handsome person:

```
.person--handsome .person__face {}
```

Here is one of a few occasions where we'd use a descendant selector to modify an Element based on a Modifier on the Block.

If using Sass, we would likely write this like so:

```
.person {}

    .person__face {

        .person--handsome & {}

    }

.person--handsome {}
```

Note that we do not nest a new instance of `.person__face {}` inside of `.person--handsome {}`; instead, we make use of Sass' parent selectors to

prepend `.person--handsome` onto the existing `.person__face {}` selector. This means that all of our `.person__face {}`-related rules exist in once place, and aren't spread throughout the file. This is general good practice when dealing with nested code: keep all of your context (e.g. all `.person__face {}` code) encapsulated in one location.

Naming Conventions in HTML

As I previously hinted at, naming conventions aren't necessarily all that useful in your CSS. Where naming conventions' power really lies is in your markup. Take the following, non-naming-conventioned HTML:

```
<div class="box  profile  pro-user">

    <img class="avatar  image" />

    <p class="bio">...</p>

</div>
```

How are the classes `box` and `profile` related to each other? How are the classes `profile` and `avatar` related to each other? Are they related at all? Should you be using `pro-user` alongside `bio`? Will the classes `image` and `profile` live in the same part of the CSS? Can you use `avatar` anywhere else?

From that markup alone, it is very hard to answer any of those questions. Using a naming convention, however, changes all that:

```
<div class="box  profile  profile--is-pro-user">

    <img class="avatar  profile__image" />
```

```
    <p class="profile__bio">...</p>

</div>
```

Now we can clearly see which classes are and are not related to each other, and how; we know what classes we can't use outside of the scope of this component; and we know which classes we may be free to reuse elsewhere.

JavaScript Hooks

As a rule, it is unwise to bind your CSS and your JS onto the same class in your HTML. This is because doing so means you can't have (or remove) one without (removing) the other. It is much cleaner, much more transparent, and much more maintainable to bind your JS onto specific classes.

I have known occasions before when trying to refactor some CSS has unwittingly removed JS functionality because the two were tied to each other—it was impossible to have one without the other.

Typically, these are classes that are prepended with `js-`, for example:

```
<input type="submit" class="btn  js-btn" value="Follow" />
```

This means that we can have an element elsewhere which can carry with style of `.btn {}`, but without the behaviour of `.js-btn`.

`data-*` Attributes

A common practice is to use `data-*` attributes as JS hooks, but this is incorrect. `data-*` attributes, as per the spec, are used '***to store custom data*** *private to the page or application*' (emphasis mine). `data-*` attributes are designed to store data, not be bound to.

Taking It Further

As previously mentioned, these are very simple naming conventions, and ones that don't do much more than denote three distinct groups of class.

I would encourage you to read up on and further look in to your naming convention in order to provide more functionality—I know it's something I'm keen to research and investigate further.

Further Reading

- ***MindBEMding – getting your head 'round BEM syntax***

CSS Selectors

Perhaps somewhat surprisingly, one of the most fundamental, critical aspects of writing maintainable and scalable CSS is selectors. Their specificity, their portability, and their reusability all have a direct impact on the mileage we will get out of our CSS, and the headaches it might bring us.

Selector Intent

It is important when writing CSS that we scope our selectors correctly, and that we're selecting the right things for the right reasons. *Selector Intent* is the process of deciding and defining what you want to style and how you will go about selecting it. For example, if you are wanting to style your website's main navigation menu, a selector like this would be incredibly unwise:

```
header ul {}
```

This selector's intent is to style any `ul` inside any `header` element, whereas *our* intent was to style the site's main navigation. This is poor Selector Intent: you can have any number of `header` elements on a page, and they in turn can house any number of `ul`s, so a selector like this runs the risk of applying very specific styling to a very wide number of elements. This will result in having to write more CSS to undo the greedy nature of such a selector.

A better approach would be a selector like:

```
.site-nav {}
```

An unambiguous, explicit selector with good Selector Intent. We are explicitly selecting the right thing for exactly the right reason.

Poor Selector Intent is one of the biggest reasons for headaches on CSS projects. Writing rules that are far too greedy—and that apply very specific treatments via very far reaching selectors—causes unexpected side effects and leads to very tangled stylesheets, with selectors overstepping their intentions and impacting and interfering with otherwise unrelated rulesets.

CSS cannot be encapsulated, it is inherently leaky, but we can mitigate some of these effects by not writing such globally-operating selectors: **your selectors should be as explicit and well reasoned as your reason for wanting to select something.**

Reusability

With a move toward a more component-based approach to constructing UIs, the idea of reusability is paramount. We want the option to be able to move, recycle, duplicate, and syndicate components across our projects.

To this end, we make heavy use of classes. IDs, as well as being hugely over-

specific, cannot be used more than once on any given page, whereas classes can be reused an infinite amount of times. Everything you choose, from the type of selector to its name, should lend itself toward being reused.

Location Independence

Given the ever-changing nature of most UI projects, and the move to more component-based architectures, it is in our interests not to style things based on where they are, but on *what* they are. That is to say, our components' styling should not be reliant upon where we place them—they should remain entirely location independent.

Let's take an example of a call-to-action button that we have chosen to style via the following selector:

```
.promo a {}
```

Not only does this have poor Selector Intent—it will greedily style any and every link inside of a `.promo` to look like a button—it is also pretty wasteful as a result of being so locationally dependent: we can't reuse that button with its correct styling outside of `.promo` because it is explicitly tied to that location. A far better selector would have been:

```
.btn {}
```

This single class can be reused anywhere outside of `.promo` and will always carry its correct styling. As a result of a better selector, this piece of UI is more portable, more recyclable, doesn't have any dependencies, and has much better Selector Intent. **A component shouldn't have to live in a certain place to look a certain way.**

Portability

Reducing, or, ideally, removing, location dependence means that we can move components around our markup more freely, but how about improving our ability to move classes around components? On a much lower level, there are changes we can make to our selectors that make the selectors themselves—as opposed to the components they create—more portable. Take the following example:

```
input.btn {}
```

This is a *qualified* selector; the leading `input` ties this ruleset to only being able to work on `input` elements. By omitting this qualification, we allow ourselves to reuse the `.btn` class on any element we choose, like an `a`, for example, or a `button`.

Qualified selectors do not lend themselves well to being reused, and every selector we write should be authored with reuse in mind.

Of course, there are times when you may want to legitimately qualify a selector—you might need to apply some very specific styling to a particular element when it carries a certain class, for example:

```
/**
 * Embolden and colour any element with a class of `.error`.
 */
.error {
    color: red;
    font-weight: bold;
}

/**
```

```
 * If the element is a `div`, also give it some box-like styling.
 */
div.error {
    padding: 10px;
    border: 1px solid;
}
```

This is one example where a qualified selector might be justifiable, but I would still recommend an approach more like:

```
/**
 * Text-level errors.
 */
.error-text {
    color: red;
    font-weight: bold;
}

/**
 * Elements that contain errors.
 */
.error-box {
    padding: 10px;
    border: 1px solid;
}
```

This means that we can apply `.error-box` to any element, and not just a `div` —it is more reusable than a qualified selector.

Quasi-Qualified Selectors

One thing that qualified selectors can be useful for is signalling where a class

might be expected or intended to be used, for example:

```
ul.nav {}
```

Here we can see that the `.nav` class is meant to be used on a `ul` element, and not on a `nav`. By using *quasi-qualified selectors* we can still provide that information without actually qualifying the selector:

```
/*ul*/.nav {}
```

By commenting out the leading element, we can still leave it to be read, but avoid qualifying and increasing the specificity of the selector.

Naming

As Phil Karlton once said, *'There are only two hard things in Computer Science: cache invalidation and naming things.'*

I won't comment on the former claim here, but the latter has plagued me for years. My advice with regard to naming things in CSS is to pick a name that is sensible, but somewhat ambiguous: aim for high reusability. For example, instead of a class like `.site-nav`, choose something like `.primary-nav`; rather than `.footer-links`, favour a class like `.sub-links`.

The differences in these names is that the first of each two examples is tied to a very specific use case: they can only be used as the site's navigation or the footer's links respectively. By using slightly more ambiguous names, we can increase our ability to reuse these components in different circumstances.

To quote Nicolas Gallagher:

> *Tying your class name semantics tightly to the nature of the content has*

already reduced the ability of your architecture to scale or be easily put to use by other developers.

That is to say, we should use sensible names—classes like `.border` or `.red` are never advisable—but we should avoid using classes which describe the exact nature of the content and/or its use cases. **Using a class name to describe content is redundant because content describes itself.**

The debate surrounding semantics has raged for years, but it is important that we adopt a more pragmatic, sensible approach to naming things in order to work more efficiently and effectively. Instead of focussing on 'semantics', look more closely at sensibility and longevity—choose names based on ease of maintenance, not for their perceived meaning.

Name things for people; they're the only things that actually *read* your classes (everything else merely matches them). Once again, it is better to strive for reusable, recyclable classes rather than writing for specific use cases. Let's take an example:

```
/**
 * Runs the risk of becoming out of date; not very maintainable.
 */
.blue {
    color: blue;
}

/**
 * Depends on location in order to be rendered properly.
 */
.header span {
    color: blue;
}
```

```
/**
 * Too specific; limits our ability to reuse.
 */
.header-color {
    color: blue;
}

/**
 * Nicely abstracted, very portable, doesn't risk becoming out of c
 */
.highlight-color {
    color: blue;
}
```

It is important to strike a balance between names that do not literally describe the style that the class brings, but also ones that do not explicitly describe specific use cases. Instead of `.home-page-panel`, choose `.masthead`; instead of `.site-nav`, favour `.primary-nav`; instead of `.btn-login`, opt for `.btn-primary`.

Naming UI Components

Naming components with agnosticism and reusability in mind really helps developers construct and modify UIs much more quickly, and with far less waste. However, it can sometimes be beneficial to provide more specific or meaningful naming alongside the more ambiguous class, particularly when several agnostic classes come together to form a more complex and specific component that might benefit from having a more meaningful name. In this scenario, we augment the classes with a `data-ui-component` attribute which houses a more specific name, for example:

```
<ul class="tabbed-nav" data-ui-component="Main Nav">
```

Here we have the benefits of a highly reusable class name which does not describe—and, therefore, tie itself to—a specific use case, and added meaning via our `data-ui-component` attribute. The `data-ui-component`'s value can take any format you wish, like title case:

```
<ul class="tabbed-nav" data-ui-component="Main Nav">
```

Or class-like:

```
<ul class="tabbed-nav" data-ui-component="main-nav">
```

Or namespaced:

```
<ul class="tabbed-nav" data-ui-component="nav-main">
```

The implementation is largely personal preference, but the concept still remains: **Add any useful or specific meaning via a mechanism that will not inhibit your and your team's ability to recycle and reuse CSS.**

Selector Performance

A topic which is—with the quality of today's browsers—more interesting than it is important, is selector performance. That is to say, how quickly a browser can match the selectors your write in CSS up with the nodes it finds in the

DOM.

Generally speaking, the longer a selector is (i.e. the more component parts) the slower it is, for example:

```
body.home div.header ul {}
```

...is a far less efficient selector than:

```
.primary-nav {}
```

This is because browsers read CSS selectors **right-to-left**. A browser will read the first selector as

- find all `ul` elements in the DOM;
- now check if they live anywhere inside an element with a class of `.header`;
- next check that `.header` class exists on a `div` element;
- now check that that all lives anywhere inside any elements with a class of `.home`;
- finally, check that `.home` exists on a `body` element.

The second, in contrast, is simply a case of the browser reading

- find all the elements with a class of `.primary-nav`.

To further compound the problem, we are using descendant selectors (e.g. `.foo .bar {}`). The upshot of this is that a browser is required to start with the rightmost part of the selector (i.e. `.bar`) and keep looking up the DOM indefinitely until it finds the next part (i.e. `.foo`). This could mean stepping up the DOM dozens of times until a match is found.

This is just one reason why **nesting with preprocessors is often a false economy**; as well as making selectors unnecessarily more specific, and creating location dependency, it also creates more work for the browser.

By using a child selector (e.g. `.foo > .bar {}`) we can make the process much more efficient, because this only requires the browser to look one level higher in the DOM, and it will stop regardless of whether or not it found a match.

The Key Selector

Because browsers read selectors right-to-left, the rightmost selector is often critical in defining a selector's performance: this is called the *key selector*.

The following selector might appear to be highly performant at first glance. It uses an ID which is nice and fast, and there can only ever be one on a page, so surely this will be a nice and speedy lookup—just find that one ID and then style everything inside of it:

```
#foo * {}
```

The problem with this selector is that the key selector (`*`) is very, *very* far reaching. What this selector actually does is find *every single* node in the DOM (even `<title>`, `<link>`, and `<head>` elements; *everything*) and then looks to see if it lives anywhere at any level within `#foo`. This is a very, *very* expensive selector, and should most likely be avoided or rewritten.

Thankfully, by writing selectors with good **Selector Intent**, we are probably avoiding inefficient selectors by default; we are very unlikely to have greedy key selectors if we're targeting the right things for the right reason.

That said, however, CSS selector performance should be fairly low on your list of things to optimise; browsers are fast, and are only ever getting faster, and it is only on notable edge cases that inefficient selectors would be likely to pose a

problem.

As well as their own specific issues, nesting, qualifying, and poor Selector Intent all contribute to less efficient selectors.

General Rules

Your selectors are fundamental to writing good CSS. To very briefly sum up the above sections:

- **Select what you want explicitly**, rather than relying on circumstance or coincidence. Good Selector Intent will rein in the reach and leak of your styles.
- **Write selectors for reusability**, so that you can work more efficiently and reduce waste and repetition.
- **Do not nest selectors unnecessarily**, because this will increase specificity and affect where else you can use your styles.
- **Do not qualify selectors unnecessarily**, as this will impact the number of different elements you can apply styles to.
- **Keep selectors as short as possible**, in order to keep specificity down and performance up.

Focussing on these points will keep your selectors a lot more sane and easy to work with on changing and long-running projects.

Further Reading

- *Shoot to kill; CSS selector intent*
- *'Scope' in CSS*
- *Keep your CSS selectors short*
- *About HTML semantics and front-end architecture*
- *Naming UI components in OOCSS*
- *Writing efficient CSS selectors*

Specificity

As we've seen, CSS isn't the most friendly of languages: globally operating, very leaky, dependent on location, hard to encapsulate, based on inheritance… But! None of that even comes close to the horrors of specificity.

No matter how well considered your naming, regardless of how perfect your source order and cascade are managed, and how well you've scoped your rulesets, just one overly-specific selector can undo everything. It is a gigantic curveball, and undermines CSS' very nature of the cascade, inheritance, and source order.

The problem with specificity is that it sets precedents and trumps that cannot *simply* be undone. If we take a real example that I was responsible for some years ago:

```
#content table {}
```

Not only does this exhibit poor **Selector Intent**—I didn't actually want every `table` in the `#content` area, I wanted a specific type of `table` that just happened to live there—it is a hugely over-specific selector. This became apparent a number of weeks later, when I needed a second type of `table`:

```
#content table {}

/**
 * Uh oh! My styles get overwritten by `#content table {}`.
 */
.my-new-table {}
```

The first selector was trumping the specificity of the one defined *after* it,

working against CSS' source-order based application of styles. In order to remedy this, I had two main options. I could

1. refactor my CSS and HTML to remove that ID;
2. write a more specific selector to override it.

Unfortunately, refactoring would have taken a long time; it was a mature product and the knock-on effects of removing this ID would have been a more substantial business cost than the second option: just write a more specific selector.

```
#content table {}

#content .my-new-table {}
```

Now I have a selector that is *even more specific still!* And if I ever want to override this one, I will need another selector of at least the same specificity defined after it. I've started on a downward spiral.

Specificity can, among other things,

- limit your ability to extend and manipulate a codebase;
- interrupt and undo CSS' cascading, inheriting nature;
- cause avoidable verbosity in your project;
- prevent things from working as expected when moved into different environments;
- lead to serious developer frustration.

All of these issues are greatly magnified when working on a larger project with a number of developers contributing code.

Keep It Low at All Times

The problem with specificity isn't necessarily that it's high or low; it's the fact it is so variant and that it cannot be opted out of: the only way to deal with it is to get progressively more specific—the notorious *specificity wars* we looked at above.

One of the single, simplest tips for an easier life when writing CSS—particularly at any reasonable scale—is to keep always try and keep specificity as low as possible at all times. Try to make sure there isn't a lot of variance between selectors in your codebase, and that all selectors strive for as low a specificity as possible.

Doing so will instantly help you tame and manage your project, meaning that no overly-specific selectors are likely to impact or affect anything of a lower specificity elsewhere. It also means you're less likely to need to fight your way out of specificity corners, and you'll probably also be writing much smaller stylesheets.

Simple changes to the way we work include, but are not limited to,

- not using IDs in your CSS;
- not nesting selectors;
- not qualifying classes;
- not chaining selectors.

Specificity can be wrangled and understood, but it is safer just to avoid it entirely.

IDs in CSS

If we want to keep specificity low, which we do, we have one really quick-win, simple, easy-to-follow rule that we can employ to help us: avoid using IDs in CSS.

Not only are IDs inherently non-reusable, they are also vastly more specific

than any other selector, and therefore become specificity anomalies. Where the rest of your selectors are relatively low specificity, your ID-based selectors are, comparatively, much, *much* higher.

In fact, to highlight the severity of this difference, see how *one thousand* chained classes cannot override the specificity of a single ID: **jsfiddle.net/0yb7rque**. (Please note that in Firefox you may see the text rendering in blue: this is a **known bug**, and an ID will be overridden by 256 chained classes.)

N.B. It is still perfectly okay to use IDs in HTML and JavaScript; it is only in CSS that they prove troublesome.

It is often suggested that developers who choose not to use IDs in CSS merely '*don't understand how specificity works*'. This is as incorrect as it is offensive: no matter how experienced a developer you are, this behaviour cannot be circumvented; no amount of knowledge will make an ID less specific.

Opting into this way of working only introduces the chance of problems occurring further down the line, and—particularly when working at scale—all efforts should be made to *avoid* the potential for problems to arise. In a sentence:

It is just not worth introducing the risk.

Nesting

We've already looked at how nesting can lead to location dependent and potentially inefficient code, but now it's time to take a look at another of its pitfalls: it makes selectors more specific.

When we talk about nesting, we don't necessarily mean preprocessor nesting, like so:

```
.foo {

    .bar {}

}
```

We're actually talking about *descendant* or *child* selectors; selectors which rely on a thing within a thing. That could look like any one of the following:

```
/**
 * An element with a class of `.bar` anywhere inside an element wit
 * `.foo`.
 */
.foo .bar {}


/**
 * An element with a class of `.module-title` directly inside an el
 * class of `.module`.
 */
.module > .module-title {}


/**
 * Any `li` element anywhere inside a `ul` element anywhere inside
 * element
 */
nav ul li {}
```

Whether you arrive at this CSS via a preprocessor or not isn't particularly important, but it is worth noting that **preprocessors tout this as a feature, where is actually to be avoided wherever possible**.

Generally speaking, each part in a compound selector adds specificity. Ergo, the fewer parts to a compound selector then the lower its overall specificity, and we always want to keep specificity low. To quote Jonathan Snook:

> *...whenever declaring your styles,* ***use the least number of selectors required to style an element.***

Let's look at an example:

```
.widget {
    padding: 10px;
}

    .widget > .widget__title {
        color: red;
    }
```

To style an element with a class of `.widget__title`, we have a selector that is twice as specific as it needs to be. That means that if we want to make any modifications to `.widget__title`, we'll need another at-least-equally specific selector:

```
.widget { ... }

    .widget > .widget__title { ... }

    .widget > .widget__title--sub {
        color: blue;
    }
```

Not only is this entirely avoidable—we caused this problem ourselves—we have a selector that is literally double the specificity it needs to be. We used

200% of the specificity actually required. And not only *that*, but this also leads to needless verbosity in our code—more to send over the wire.

As a rule, **if a selector will work without it being nested then do not nest it**.

Scope

One possible advantage of nesting—which, unfortunately, does not outweigh the disadvantages of increased specificity—is that it provides us with a namespace of sorts. A selector like `.widget .title` scopes the styling of `.title` to an element that only exists inside of an element carrying a class of `.widget`.

This goes some way to providing our CSS with scope and encapsulation, but does still mean that our selectors are twice as specific as they need to be. A better way of providing this scope would be via a namespace—which we already have in the form of **BEM-like Naming**—which does not lead to an unnecessary increase in specificity.

Now we have better scoped CSS with minimal specificity—the best of both worlds.

Further Reading

- ***'Scope' in CSS***

`!important`

The word `!important` sends shivers down the spines of almost all front-end developers. `!important` is a direct manifestation of problems with specificity; it is a way of cheating your way out of specificity wars, but usually comes at a heavy price. It is often viewed as a last resort—a desperate, defeated stab at patching over the symptoms of a much bigger problem with your code.

The general rule is that `!important` is always a bad thing, but, to quote Jamie Mason:

> *Rules are the children of principles.*

That is to say, a single rule is a simple, black-and-white way of adhering to a much larger principle. When you're starting out, the rule '*never use* `!important`' is a good one.

However, once you begin to grow and mature as a developer, you begin to understand that the principle behind that rule is simply about keeping specificity low. You'll also learn when and where the rules can be bent…

`!important` does have a place in CSS projects, but only if used sparingly and proactively.

Proactive use of `!important` is when it is used *before* you've encountered any specificity problems; when it is used as a guarantee rather than as a fix. For example:

```
.one-half {
    width: 50% !important;
}

.hidden {
    display: none !important;
}
```

These two helper, or *utility*, classes are very specific in their intentions: you would only use them if you wanted something to be rendered at 50% width or not rendered at all. If you didn't want this behaviour, you would not use these classes, therefore whenever you do use them you will definitely want them to win.

Here we proactively apply `!important` to ensure that these styles always win. This is correct use of `!important` to guarantee that these trumps always work, and don't accidentally get overridden by something else more specific.

Incorrect, *reactive* use of `!important` is when it is used to combat specificity problems after the fact: applying `!important` to declarations because of poorly architected CSS. For example, let's imagine we have this HTML:

```
<div class="content">
    <h2 class="heading-sub">...</h2>
</div>
```

...and this CSS:

```
.content h2 {
    font-size: 2em;
}

.heading-sub {
    font-size: 1.5em !important;
}
```

Here we can see how we've used `!important` to force our `.heading-sub {}` styles to reactively override our `.content h2 {}` selector. This could have been circumvented by any number of things, including using better Selector Intent, or avoiding nesting.

In these situations, it is preferable that you investigate and refactor any offending rulesets to try and bring specificity down across the board, as opposed to introducing such specificity heavyweights.

Only use `!important` proactively, not reactively.

Hacking Specificity

With all that said on the topic of specificity, and keeping it low, it is inevitable that we will encounter problems. No matter how hard we try, and how conscientious we are, there will always be times that we need to hack and wrangle specificity.

When these situations do arise, it is important that we handle the hacks as safely and elegantly as possible.

In the event that you need to increase the specificity of a class selector, there are a number of options. We could nest the class inside something else to bring its specificity up. For example, we could use `.header .site-nav {}` to bring up the specificity of a simple `.site-nav {}` selector.

The problem with this, as we've discussed, is that it introduces location dependency: these styles will only work when the `.site-nav` component is in the `.header` component.

Instead, we can use a much safer hack that will not impact this component's portability: we can chain that class with itself:

```
.site-nav.site-nav {}
```

This chaining doubles the specificity of the selector, but does not introduce any dependency on location.

In the event that we do, for whatever reason, have an ID in our markup that we cannot replace with a class, select it via an attribute selector as opposed to an ID selector. For example, let's imagine we have embedded a third-party widget on our page. We can style the widget via the markup that it outputs, but we have no ability to edit that markup ourselves:

```
<div id="third-party-widget">
    ...
</div>
```

Even though we know not to use IDs in CSS, what other option do we have? We want to style this HTML but have no access to it, and all it has on it is an ID.

We do this:

```
[id="third-party-widget"] {}
```

Here we are selecting based on an attribute rather than an ID, and attribute selectors have the same specificity as a class. This allows us to style based on an ID, but without introducing its specificity.

Do keep in mind that these *are* hacks, and should not be used unless you have no better alternative.

Further Reading

- ***Hacks for dealing with specificity***

Architectural Principles

You would be forgiven for thinking that an architecture for CSS is a somewhat grandiose and unnecessary concept: why would something so simple, so *straightforward*, need something as complex or considered as an architecture?!

Well, as we've seen, CSS' simplicity, its looseness, and its unruly nature mean that the best way of managing (read, taming) it at any reasonable scale is through a strict and specific architecture. A solid architecture can help us control our specificity, enforce naming conventions, manage our source order, create a sane development environment, and generally make managing our CSS projects a lot more consistent and comfortable.

There is no tool, no preprocessor, no magic bullet, that will make your CSS better on its own: a developer's best tool when working with such a loose syntax is self-discipline, conscientiousness, and diligence, and a well-defined architecture will help enforce and facilitate these traits.

Architectures are large, overarching, principle-led collections of smaller conventions which come together to provide a managed environment in which code is written and maintained. Architectures are typically quite high level, and leave implementation details—such as naming conventions or syntax and formatting, for example—to the team implementing it.

Most architectures are usually based around existing design patterns and paradigms, and, more often than not, these paradigms were born of computer scientists and software engineers. For all CSS isn't 'code', and doesn't exhibit many traits that programming languages do, we find that we can apply some of these same principles to our own work.

In this section, we'll take a look at some of these design patterns and paradigms, and how we can use them to reduce code—and increase code reuse—in our CSS projects.

High-level Overview

At a very high-level, your architecture should help you

- provide a consistent and sane environment;
- accommodate change;

- grow and scale your codebase;
- promote reuse and efficiency;
- increase productivity.

Typically, this will mean a class-based and componentised architecture, split up into manageable modules, probably using a preprocessor. Of course, there is far more to an architecture than that, so let's look at some principles...

Object-orientation

Object-orientation is a programming paradigm that breaks larger programs up into smaller, in(ter)dependent objects that all have their own roles and responsibilities. From Wikipedia:

> *Object-oriented programming (OOP) is a programming paradigm that represents the concept of 'objects' [...] which are usually instances of classes, [and] are used to interact with one another to design applications and computer programs.*

When applied to CSS, we call it object-oriented CSS, or *OOCSS*. OOCSS was coined and popularised by Nicole Sullivan, whose *Media Object* has become the poster child of the methodology.

OOCSS deals with the separation of UIs into *structure* and *skin*: breaking UI components into their underlying structural forms, and layering their cosmetic forms on separately. This means that we can recycle common and recurring design *patterns* very cheaply without having to necessarily recycle their specific implementation details at the same time. OOCSS promotes reuse of code, which makes us quicker, as well as keeping the size of our codebase down.

Structural aspects can be thought of like skeletons; common, recurring frames that provide design-free constructs known as *objects* and *abstractions*. Objects and abstractions are simple design patterns that are devoid of any cosmetics; we abstract out the shared structural traits from a series of components into a

generic object.

Skin is a layer that we (optionally) add to our structure in order to give objects and abstractions a specific look-and-feel. Let's look at an example:

```
/**
 * A simple, design-free button object. Extend this object with a `
 * class.
 */
.btn {
    display: inline-block;
    padding: 1em 2em;
    vertical-align: middle;
}


/**
 * Positive buttons' skin. Extends `.btn`.
 */
.btn--positive {
    background-color: green;
    color: white;
}

/**
 * Negative buttons' skin. Extends `.btn`.
 */
.btn--negative {
    background-color: red;
    color: white;
}
```

Above, we can see how the `.btn {}` class simply provides structural styling to

an element, and doesn't concern itself with any cosmetics. We supplement the `.btn {}` object with a second class, such as `.btn--negative {}` in order to give that DOM node specific cosmetics:

```
<button class="btn  btn--negative">Delete</button>
```

Favour the multiple-class approach over using something like `@extend`: using multiple classes in your markup—as opposed to wrapping the classes up into one using a preprocessor—

- gives you a better paper-trail in your markup, and allows you to see quickly and explicitly which classes are acting on a piece of HTML;
- allows for greater composition in that classes are not tightly bound to other styles in your CSS.

Whenever you are building a UI component, try and see if you can break it into two parts: one for structural styles (paddings, layout, etc.) and another for skin (colours, typefaces, etc.).

Further Reading

- **The media object saves hundreds of lines of code**
- **The flag object**
- **Naming UI components in OOCSS**

The Single Responsibility Principle

The *single responsibility principle* is a paradigm that, very loosely, states that all pieces of code (in our case, classes) should focus on doing one thing and one thing only. More formally:

> *...the single responsibility principle states that every context (class, function, variable, etc.) should have a single responsibility, and that responsibility*

should be entirely encapsulated by the context.

What this means for us is that our CSS should be composed of a series of much smaller classes that focus on providing very specific and limited functionality. This means that we need to decompose UIs into their smallest component pieces that each serve a single responsibility; they all do just one job, but can be very easily combined and composed to make much more versatile and complex constructs. Let's take some example CSS that does not adhere to the single responsibility principle:

```
.error-message {
    display: block;
    padding: 10px;
    border-top: 1px solid #f00;
    border-bottom: 1px solid #f00;
    background-color: #fee;
    color: #f00;
    font-weight: bold;
}

.success-message {
    display: block;
    padding: 10px;
    border-top: 1px solid #0f0;
    border-bottom: 1px solid #0f0;
    background-color: #efe;
    color: #0f0;
    font-weight: bold;
}
```

Here we can see that—despite being named after one very specific use-case—these classes are handling quite a lot: layout, structure, and cosmetics. We also have a *lot* of repetition. We need to refactor this in order to abstract out some

shared objects (OOCSS) and bring it more inline with the single responsibility principle. We can break these two classes out into four much smaller responsibilities:

```
.box {
    display: block;
    padding: 10px;
}


.message {
    border-style: solid;
    border-width: 1px 0;
    font-weight: bold;
}

.message--error {
    background-color: #fee;
    color: #f00;
}

.message--success {
    background-color: #efe;
    color: #0f0;
}
```

Now we have a general abstraction for boxes which can live, and be used, completely separately from our message component, and we have a base message component that can be extended by a number of smaller responsibility classes. The amount of repetition has been greatly reduced, and our ability to extend and compose our CSS has been greatly increased. This is a great example of OOCSS and the single responsibility principle working in tandem.

By focussing on single responsibilities, we can give our code much more flexibility, and extending components' functions becomes very simple when sticking to the *open/closed principle*, which we're going to look at next.

Further Reading

- **The single responsibility principle applied to CSS**

The Open/Closed Principle

The *open/closed principle*, in my opinion, is rather poorly named. It is poorly named because 50% of the vital information is omitted from its title. The open/closed principle states that

> *[s]oftware entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*

See? The most important words—*extension* and *modification*—are completely missing from the name, which isn't very useful at all.

Once you have trained yourself to remember what the words open and closed actually relate to, you'll find that open/closed principle remarkably simple: any additions, new functionality, or features we add to our classes should be added via *extension*—we should not modify these classes directly. This really trains us to write bulletproof single responsibilities: because we shouldn't modify objects and abstractions directly, we need to make sure we get them as simple as possible the first time. This means that we should never need to actually change an abstraction—we'd simply stop using it—but any slight variants of it can be made very easily by extending it.

Let's take an example:

```
.box {
```

```
    display: block;
    padding: 10px;
}

.box--large {
    padding: 20px;
}
```

Here we can see that the `.box {}` object is incredibly simple: we've stripped it right back into one very small and very focussed responsibility. To modify that box, we extend it with another class; `.box--large {}`. Here the `.box {}` class is closed to modification, but open to being extended.

An incorrect way of achieving the same might look like this:

```
.box {
    display: block;
    padding: 10px;
}

.content .box {
    padding: 20px;
}
```

Not only is this overly specific, locationally dependent, and potentially displaying poor Selector Intent, we are modifying the `.box {}` directly. We should rarely—if ever—find an object or abstraction's class as a key selector in a compound selector.

A selector like `.content .box {}` is potentially troublesome because

- it forces all `.box` components into that style when placed inside of

`.content`, which means the modification is dictated to developers, whereas developers should be allowed to opt into changes explicitly;
- the `.box` style is now unpredictable to developers; the single responsibility no longer exists because nesting the selector produces a forced caveat.

All modifications, additions, and changes should always be opt-in, not mandatory. If you think something might need a slight adjustment to take it away from the norm, provide another class which adds this functionality.

When working in a team environment, be sure to write API-like CSS; always ensure that existing classes remain backward compatible (i.e. no changes at their root) and provide new hooks to bring in new features. Changing the root object, abstraction, or component could have huge knock-on effects for developers making use of that code elsewhere, so never modify existing code directly.

Exceptions may present themselves when it transpires that a root object does need a rewrite or refactor, but it is only in these specific cases that you should modify code. Remember: **open for extension; closed for modification**.

Further Reading

- **The open/closed principle applied to CSS**

DRY

DRY, which stands for *Don't Repeat Repeat Yourself*, is a micro-principle used in software development which aims to keep the repetition of key information to a minimum. Its formal definition is that

> *[e]very piece of knowledge must have a single, unambiguous, authoritative representation within a system.*

Although a very simple principle—in principle—DRY is often misinterpreted as the necessity to never repeat the exact same thing twice at all in a project. This is impractical and usually counterproductive, and can lead to forced abstractions, over-thought and -engineered code, and unusual dependencies.

The key isn't to avoid all repetition, but to normalise and abstract *meaningful* repetition. If two things happen to share the same declarations coincidentally, then we needn't DRY anything out; that repetition is purely circumstantial and cannot be shared or abstracted. For example:

```
.btn {
    display: inline-block;
    padding: 1em 2em;
    font-weight: bold;
}

[...]

.page-title {
    font-size: 3rem;
    line-height: 1.4;
    font-weight: bold;
}

[...]

    .user-profile__title {
        font-size: 1.2rem;
        line-height: 1.5;
        font-weight: bold;
    }
```

From the above code, we can reasonably deduce that the `font-weight: bold;`

declaration appears three times purely coincidentally. To try and create an abstraction, mixin, or `@extend` directive to cater for this repetition would be overkill, and would tie these three rulesets together based purely on circumstance.

However, imagine we're using a web-font that requires `font-weight: bold;` to be declared every time the `font-family` is:

```
.btn {
    display: inline-block;
    padding: 1em 2em;
    font-family: "My Web Font", sans-serif;
    font-weight: bold;
}

[...]

.page-title {
    font-size: 3rem;
    line-height: 1.4;
    font-family: "My Web Font", sans-serif;
    font-weight: bold;
}

[...]

    .user-profile__title {
        font-size: 1.2rem;
        line-height: 1.5;
        font-family: "My Web Font", sans-serif;
        font-weight: bold;
    }
```

Here we're repeating a more meaningful snippet of CSS; these two declarations have to always be declared together. In this instance, we probably would DRY out our CSS.

I would recommend using a mixin over `@extend` here because, even though the two declarations are thematically grouped, the rulesets themselves are still separate, unrelated entities: to use `@extend` would be to physically group these unrelated rulesets together in our CSS, thus making the unrelated related.

Our mixin:

```
@mixin my-web-font() {
    font-family: "My Web Font", sans-serif;
    font-weight: bold;
}

.btn {
    display: inline-block;
    padding: 1em 2em;
    @include my-web-font();
}

[...]

.page-title {
    font-size: 3rem;
    line-height: 1.4;
    @include my-web-font();
}

[...]

    .user-profile__title {
```

```
        font-size: 1.2rem;
        line-height: 1.5;
        @include my-web-font();
    }
```

Now the two declarations only exist once, meaning we're not repeating ourselves. If we ever switch out our web-font, or move to a `font-weight: normal;` version, we only need to make that change in one place.

In short, only DRY code that is actually, thematically related. Do not try to reduce purely coincidental repetition: **duplication is better than the wrong abstraction**.

Further Reading

- **Writing DRYer vanilla CSS**

Composition over Inheritance

Now that we're used to spotting abstractions and creating single responsibilities, we should be in a great position to start composing more complex composites from a series of much smaller component parts. Nicole Sullivan likens this to using Lego; tiny, single responsibility pieces which can be combined in a number of different quantities and permutations to create a multitude of very different looking results.

This idea of building through composition is not a new one, and is often referred to as *composition over inheritance*. This principle suggests that larger systems should be composed from much smaller, individual parts, rather than inheriting behaviour from a much larger, monolithic object. This should keep your code decoupled—nothing inherently relies on anything else.

Composition is a very valuable principle for an architecture to make use of,

particularly considering the move toward component-based UIs. It will mean you can more easily recycle and reuse functionality, as well rapidly construct larger parts of UI from a known set of composable objects. Think back to our error message example in the **Single Responsibility Principle** section; we created a complete UI component by composing a number of much smaller and unrelated objects.

The Separation of Concerns

The *separation of concerns* is a principle which, at first, sounds a lot like the single responsibility principle. The separation of concerns states that code should be broken up

> *into distinct sections, such that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program. [...] A program that embodies SoC well is called a modular program.*

Modular is a word we're probably used to; the idea of breaking UIs and CSS into much smaller, composable pieces. The separation of concerns is just a formal definition which covers the concepts of modularity and encapsulation in code. In CSS this means building individual components, and writing code which only focusses itself on one task at a time.

The term was coined by Edsger W. Dijkstra, who rather elegantly said:

> *Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained—on the contrary!—by tackling these various aspects simultaneously. It is what I sometimes have called 'the*

> *separation of concerns', which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by 'focusing one's attention upon some aspect': it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.*

Beautiful. The idea here is to focus fully on one thing at once; build one thing to do its job very well whilst paying as little attention as possible to other facets of your code. Once you have addressed and built all these separate concerns in isolation—meaning they're probably very modular, decoupled, and encapsulated—you can begin bringing them together into a larger project.

A great example is layout. If you are using a grid system, all of the code pertaining to layout should exist on its own, without including anything else. You've written code that handles layout, and that's it:

```
<div class="layout">

    <div class="layout__item  two-thirds">
    </div>

    <div class="layout__item  one-third">
    </div>

</div>
```

You will now need to write new, separate code to handle what lives inside of that layout:

```
<div class="layout">
```

```
    <div class="layout__item  two-thirds">
        <section class="content">
            ...
        </section>
    </div>

    <div class="layout__item  one-third">
        <section class="sub-content">
            ...
        </section>
    </div>

</div>
```

The separation of concerns allows you to keep code self-sufficient, ignorant, and ultimately a lot more maintainable. Code which adheres to the separation of concerns can be much more confidently modified, edited, extended, and maintained because we know how far its responsibilities reach. We know that modifying layout, for example, will only ever modify layout—nothing else.

The separation of concerns increases reusability and confidence whilst reducing dependency.

Misconceptions

There are, I feel, a number of unfortunate misconceptions surrounding the separation of concerns when applied to HTML and CSS. They all seem to revolve around some format of:

> *Using classes for CSS in your markup breaks the separation of concerns.*

Unfortunately, this is simply not true. The separation of concerns *does* exist in the context of HTML and CSS (and JS), but not in the way a lot of people seem

to believe.

The separation of concerns when applied to front-end code is not about classes-in-HTML-purely-for-styling-hooks-blurring-the-lines-between-concerns; it is about the very fact that we are using different languages for markup and styling at all.

Back before CSS was widely adopted, we'd use `table`s to lay content out, and `font` elements with `color` attributes to provide cosmetic styling. The problem here is that HTML was being used to create content and also to style it; there was no way of having one without the other. This was a complete lack of separation of concerns, which was a problem. CSS' job was to provide a completely new syntax to apply this styling, allowing us to separate our content and style concerns across two technologies.

Another common argument is that *'putting classes in your HTML puts style information in your markup'*.

So, in a bid to circumvent this, people adopt selectors that might look a little like this:

```
body > header:first-of-type > nav > ul > li > a {
}
```

This CSS—presumably to style our site's main nav—has the usual problems of location dependency, poor Selector Intent, and high specificity, but it also manages to do *exactly what developers are trying to avoid*, only in the opposite direction: **it puts DOM information in your CSS**. Aggressive attempts to avoid putting any style hints or hooks in markup only lead to overloading stylesheets with DOM information.

In short: having classes in your markup does not violate the separation of concerns. Classes merely act as an API to link two separate concerns together.

The simplest way to separate concerns is to write well formed HTML and well formed CSS, and link to two together with sensible, judicious use of classes.

2.2.4

Last updated: 25 June, 2015