CSS Standards

**INTRODUCTION**:

This document describes the CSS coding standards that need to be followed at the time of implementation.

Part I covers only the css rules. Part II will be covering the CSS Best Practices.

## Part I - CSS Style Rules

**CSS Validity**

Use valid CSS where possible.

Unless dealing with CSS validator bugs or requiring proprietary syntax, use valid CSS code.

Use tools such as the W3C CSS validator to test.

Using valid CSS is a measurable baseline quality attribute that allows to spot CSS code that may not have any effect and can be removed, and that ensures proper CSS usage.

### ID and Class Naming

Use meaningful or generic ID and class names.

Instead of presentational or cryptic names, always use ID and class names that reflect the purpose of the element in question, or that are otherwise generic.

Names that are specific and reflect the purpose of the element should be preferred as these are most understandable and the least likely to change.

Generic names are simply a fallback for elements that have no particular or no meaning different from their siblings. They are typically needed as “helpers.”

Using functional or generic names reduces the probability of unnecessary document or template changes.

/\* Not recommended: meaningless \*/

#yee-1901 {}

/\* Not recommended: presentational \*/

.button-green {}

.clear {}

/\* Recommended: specific \*/

#gallery {}

#login {}

.video {}

/\* Recommended: generic \*/

.aux {}

.alt {}

### ID and Class Name Style

Use ID and class names that are as short as possible but as long as necessary.

Try to convey what an ID or class is about while being as brief as possible.

Using ID and class names this way contributes to acceptable levels of understandability and code efficiency.

/\* Not recommended \*/

#navigation {}

.atr {}

/\* Recommended \*/

#nav {}

.author {}

### Type Selectors

Avoid qualifying ID and class names with type selectors.

Unless necessary (for example with helper classes), do not use element names in conjunction with IDs or classes.

Avoiding unnecessary ancestor selectors is useful for performance reasons.

/\* Not recommended \*/

ul#example {}

div.error {}

/\* Recommended \*/

#example {}

.error {}

### Shorthand Properties

Use shorthand properties where possible.

CSS offers a variety of shorthand properties (like font) that should be used whenever possible, even in cases where only one value is explicitly set.

Using shorthand properties is useful for code efficiency and understandability.

/\* Not recommended \*/

border-top-style: none;

font-family: palatino, georgia, serif;

font-size: 100%;

line-height: 1.6;

padding-bottom: 2em;

padding-left: 1em;

padding-right: 1em;

padding-top: 0;

/\* Recommended \*/

border-top: 0;

font: 100%/1.6 palatino, georgia, serif;

padding: 0 1em 2em;

### 0 and Units

Omit unit specification after “0” values.

Do not use units after 0 values unless they are required.

margin: 0;

padding: 0;

### Leading 0s

Omit leading “0”s in values.

Do not use put 0s in front of values or lengths between -1 and 1.

font-size: .8em;

### Hexadecimal Notation

Use 3 character hexadecimal notation where possible.

For color values that permit it, 3 character hexadecimal notation is shorter and more succinct.

/\* Not recommended \*/

color: #eebbcc;

/\* Recommended \*/

color: #ebc;

### Prefixes

Prefix selectors with an application-specific prefix (optional).

In large projects as well as for code that gets embedded in other projects or on external sites use prefixes (as namespaces) for ID and class names. Use short, unique identifiers followed by a dash.

Using namespaces helps preventing naming conflicts and can make maintenance easier, for example in search and replace operations.

.adw-help {} /\* AdWords \*/

#maia-note {} /\* Maia \*/

### ID and Class Name Delimiters

Separate words in ID and class names by a hyphen.

Do not concatenate words and abbreviations in selectors by any characters (including none at all) other than hyphens, in order to improve understanding and scannability.

/\* Not recommended: does not separate the words “demo” and “image” \*/

.demoimage {}

/\* Not recommended: uses underscore instead of hyphen \*/

.error\_status {}

/\* Recommended \*/

#video-id {}

.ads-sample {}

### Hacks

Avoid user agent detection as well as CSS “hacks”—try a different approach first.

It’s tempting to address styling differences over user agent detection or special CSS filters, workarounds, and hacks. Both approaches should be considered last resort in order to achieve and maintain an efficient and manageable code base. Put another way, giving detection and hacks a free pass will hurt projects in the long run as projects tend to take the way of least resistance. That is, allowing and making it easy to use detection and hacks means using detection and hacks more frequently—and more frequently is too frequently.

## CSS Formatting Rules

### Declaration Order

Alphabetize declarations.

Put declarations in alphabetical order in order to achieve consistent code in a way that is easy to remember and maintain.

Ignore vendor-specific prefixes for sorting purposes. However, multiple vendor-specific prefixes for a certain CSS property should be kept sorted (e.g. -moz prefix comes before -webkit).

background: fuchsia;

border: 1px solid;

-moz-border-radius: 4px;

-webkit-border-radius: 4px;

border-radius: 4px;

color: black;

text-align: center;

text-indent: 2em;

### Block Content Indentation

Indent all block content.

Indent all [block content](http://www.w3.org/TR/CSS21/syndata.html#block), that is rules within rules as well as declarations, so to reflect hierarchy and improve understanding.

@media screen, projection {

html {

background: #fff;

color: #444;

}

}

### Declaration Stops

Use a semicolon after every declaration.

End every declaration with a semicolon for consistency and extensibility reasons.

/\* Not recommended \*/

.test {

display: block;

height: 100px

}

/\* Recommended \*/

.test {

display: block;

height: 100px;

}

### Property Name Stops

Use a space after a property name’s colon.

Always use a single space between property and value (but no space between property and colon) for consistency reasons.

/\* Not recommended \*/

h3 {

font-weight:bold;

}

/\* Recommended \*/

h3 {

font-weight: bold;

}

### Declaration Block Separation

Use a space between the last selector and the declaration block.

Always use a single space between the last selector and the opening brace that begins the declaration block.

The opening brace should be on the same line as the last selector in a given rule.

/\* Not recommended: missing space \*/

#video{

margin-top: 1em;

}

/\* Not recommended: unnecessary line break \*/

#video

{

margin-top: 1em;

}

/\* Recommended \*/

#video {

margin-top: 1em;

}

### Selector and Declaration Separation

Separate selectors and declarations by new lines.

Always start a new line for each selector and declaration.

/\* Not recommended \*/

a:focus, a:active {

position: relative; top: 1px;

}

/\* Recommended \*/

h1,

h2,

h3 {

font-weight: normal;

line-height: 1.2;

}

### Rule Separation

Separate rules by new lines.

Always put a blank line (two line breaks) between rules.

html {

background: #fff;

}

body {

margin: auto;

width: 50%;

}

### CSS Quotation Marks

Use single quotation marks for attribute selectors and property values.

Use single ('') rather than double ("") quotation marks for attribute selectors or property values. Do not use quotation marks in URI values (url()).

Exception: If you do need to use the @charset rule, use double quotation marks—single quotation marks are not permitted.

/\* Not recommended \*/

@import url("//www.google.com/css/maia.css");

html {

font-family: "open sans", arial, sans-serif;

}

/\* Recommended \*/

@import url(//www.google.com/css/maia.css);

html {

font-family: 'open sans', arial, sans-serif;

}

## CSS Meta Rules

### Section Comments

Group sections by a section comment (optional).

If possible, group style sheet sections together by using comments. Separate sections with new lines.

/\* Header \*/

#adw-header {}

/\* Footer \*/

#adw-footer {}

/\* Gallery \*/

.adw-gallery {}

**Part II – CSS Best Practices and Page Architecture techniques.**

**Syntax and Formatting**

At a very high-level, we want

* Four (4) space indents. No tabs;
* 80 character wide columns;
* Multi-line CSS;
* Meaningful use of whitespace.

Multiple Files

It is a good idea to split discrete chunks of code into their own files, which are concatenated during a build step.

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.

Example

/\*\*

\* I am a long-form comment. I describe, in detail, the CSS that follows. I am

\* such a long comment that I easily break the 80 character limit, so I am

\* broken across several lines.

\*/

Titling

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

Example:

/\*------------------------------------\*\

#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.

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 diffs, 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; }

Indenting

As well as intending 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.

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 {}

Naming Conventions

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.

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 {}

Hybrid Naming

The naming of the classes will be following will include BEM based and Atomic based.

BEM Based Naming:

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.

**Note**: BEM naming only to be applied to appearance (e.g. colors, borders, backgrounds, etc).

Atomic Naming:

This generally should only be applied to properties like spacing and layouts. In Atomic naming one would generally break down the names in the form of abbrevations.

For Example:

Semantic CSS:

.nav-link {

float: left;

margin-right: 16px;

color: white;

}

.nav-link-selected {

color: teal;

}

The above css styles can be written in a more granular level for portability and reusability purpose like

Atomic CSS:

.fl-st { float: left; }

.mr-1 { margin-right: 16px; }

.c-w { color: white; }

.c-pr { color: teal; }

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.

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 uls, 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.

Reusability

Location Independence

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 {}

ot only does this have poor Selector Intent—it will style any and every link inside of a .promo to look like a button. 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.

Naming

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.

*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.*

*By Nicolas Gallagher*

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.

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 date.

\*/

.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.

Selector Performance

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.

When it comes to descendant selectors (e.g .foo .bar {} ). It is painful for browsers as it has to search for the right most part and go up the DOM indefinitely until it finds the parent selector.

A better solution would be to use a child selector( e.g .foo > .bar {} ).

This way the process is much more efficient.

Specificity

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.