MaintainableCSS

Write CSS without worrying that overzealous, pre-existing styles will cause problems. MaintainableCSS is an approach to writing modular, scalable and maintainable CSS.

“A handy little read on learning how to write modular and maintainable CSS.”
Smashing Magazine

“Finally a good book on how to write maintainable CSS.”
Alexander Dajani

“I actually love everything about this.”
Simon Taggart

START READING

Written by Adam Silver, an interaction designer from London, UK. Contribute on GitHub.

CHAPTER 1

Introduction

MaintainableCSS is an approach to writing modular, scalable and maintainable CSS for small and large codebases. You can learn it in 20 minutes and implement it immediately in your project.

Modular

A module is a distinct, independent unit that can be combined with other modules to create a more complex structure. In a living room, we can consider the TV, the sofa and the wall art to be modules.

When one module is taken away, the other modules continue to work. For example, we can sit on the sofa even if the TV breaks down. On a web page, a header, product list and menu can all be thought about as modules.

Scalable

Scalable CSS means that as CSS increases in size, it's still easy to maintain. If you've ever inherited a large CSS codebase, and been afraid to make changes, you'll sympathise with this.

Maintainable

Maintainable CSS makes it easy to make styling changes without worrying about accidentally causing problems elsewhere.

NEXT: SEMANTICS

CHAPTERS

Written by Adam Silver, an interaction designer from London, UK. Contribute on GitHub.

CHAPTER 2

Semantics

Semantic HTML isn't only about the elements we use. It's quite obvious that we should use `<a>` for links, `<table>` for tabular data and `<p>` for paragraphs etc. What's less obvious is the names we use for classes.

As Phil Karton says, “there are only two hard things in Computer Science: cache invalidation and **naming things**.” So spending a whole chapter talking about naming is essential.

Naming is the most important aspect of writing maintainable CSS. There are two main approaches: the semantic approach and the non-semantic approach.

Semantic vs non-semantic

```
<!-- non semantic -->
<div class="red pull-left pb3">
<div class="grid row">
<div class="col-xs-4">

<!-- semantic -->
<div class="basket">
<div class="product">
<div class="searchResults">
```

Non-semantic classes don't convey *what* an element represents. At most they give you an idea of what an element *looks* like. Atomic, visual, behavioural and utility classes are all forms of non-semantic classes.

Semantic classes don't convey their styles, but that's fine—that's what CSS is for. Semantic classes mean something to HTML, CSS, Javascript and automated functional tests.

Let's look at why semantic classes usually work best.

1. Because they're readable

Here's a real snippet of HTML using atomic classes:

```
<div class="pb3 pb4-ns pt4 pt5-ns mt4 black-70 fl-l
  <h1 class="f4 fw6 f1-ns lh-title measure mt0">Heac
  <p class="f5 f4-ns fw4 b measure dib-m lh-copy">Ta
</div>
```

- Words are generally easier to understand than abbreviations which have to be understood and interpreted before knowing what they stand for
- It's unclear where the module begins and ends
- We have to wade through a very long list of classes to work out what's going on; which classes override which; and which apply at different breakpoints
- The classes lack clarity—for example, it's not clear whether `black-70` is referring to the foreground or background colour
- The content is obfuscated by the larger surrounding HTML

- The HTML is large in size

Here's the same HTML with semantic classes:

```
<div class="hero">
  <h1 class="hero-title">Heading</h1>
  <p class="hero-tagline">Tagline</p>
</div>
```

- These classes are easy to read without needing to be interpreted
- It's clear where the module begins and ends
- It's easy to read the CSS because it uses the standard syntax for this purpose
- The content is no longer obfuscated
- The HTML is half the size

2. Because they make it easier to build responsive sites

Imagine coding a two-column responsive grid whereby:

- each column has `20px` and `50px` padding on small and large screens respectively
- each column has `2em` and `3em` font-size on small and large screens respectively
- the columns stack on small screens. Note that *column* is now a misleading class name.

With non-semantic classes it may look like this:

```
<div class="grid clearfix">
  <div class="col pd20 pd50 fs2 fs3">Column 1</div>
  <div class="col pd20 pd50 fs2 fs3">Column 2</div>
</div>
```

- There are 7 classes—some of which override each other
- To make the columns work responsively we would need a `fs3large` class which means creating a naming convention that recreates language constructs already provided by CSS
- At certain break points, the classes are misleading and redundant—for example `.clearfix` doesn't clear on small screens

With semantic classes it looks like this:

```
<div class="thing">
  <div class="thing-thingA"></div>
  <div class="thing-thingB"></div>
</div>
```

- The classes are encapsulated to the module's design and content
- It's easy to style elements without having to write a multiple classes and changing the HTML
- The classes are meaningful in small and big screens
- Media queries can be used to clear elements when needed

Question: how valuable is a codified responsive grid system? A layout should adapt to the content, not the other way around.

3. Because they're easier to find

Searching HTML with non-semantic classes yields many results. Searching HTML with semantic classes should yield one result which makes it quick to track down.

4. Because they reduce the chance of regression

Updating a non-semantic class could cause regression across multiple elements. Updating a semantic class only applies to the specific module, eliminating the risk of regression.

5. Because visual classes aren't necessarily valuable

Atomic CSS and inline CSS aren't totally equivalent. For example, inline CSS can't use media queries and styling in HTML stops us from being able to cache it.

But, in some ways we may as well use inline styles—at least it's explicit and reduces the CSS footprint to zero.

Question: isn't `.red` the exact same abstraction that CSS already gives us for free with `color: red`?

6. Because they provide hooks for automated tests

Automated functional tests work by searching for and interacting with elements. For example, a test might involve:

1. clicking a link

2. finding a text box
3. typing in text
4. submitting a form
5. verifying some criteria

We can't use non-semantic classes to target specific elements. And adding hooks specifically for tests is wasteful as the user now has to download extra code.

7. Because they provide hooks for JavaScript

We can't use non-semantic classes to target specific elements in order to enhance them with JavaScript.

8. Because they need less maintaining

When you name an element based on what it is, you don't have to update the HTML. That's because, a heading, for example, is always a heading, no matter what it looks like.

With visual classes, both the HTML and the CSS need updating —assuming there aren't any selectors available for use.

9. Because they're easier to debug

Inspecting an element with multiple atomic classes, means having to wade through many selectors. With a semantic class, there's only one which is much easier to work with.

10. Because the standards recommend it

On using the class attribute, HTML5 specs say in 3.2.5.7:

"[...] authors are encouraged to use values that describe the nature of the content, rather than values that describe the desired presentation of the content."

11. Because styling state is easier

Consider this HTML:

```
<a class="padding-left-20 red" href="#"></a>
```

Changing the padding and colour on hover is a difficult task. Try to avoid having to fix self-induced problems like this.

12. Because they produce a small HTML footprint

Atomic classes create extra code in HTML. Semantic classes result in less code. And while the CSS may increase in size, it can be cached.

Final thought

Semantic classes are a cornerstone of MaintainableCSS. Without them, everything else makes little sense. Name something based on what it is and everything else falls into place.

NEXT: REUSE

CHAPTERS

Written by Adam Silver, an interaction designer from London, UK. Contribute on GitHub.

CHAPTER 3

Reuse

As Harry Roberts says, “DRY is often misinterpreted as the necessity to never repeat the exact same thing twice. This is impractical and usually counterproductive, and can lead to forced abstractions, over-thought and over-engineered code.”

This forced abstraction, over-thought and over-engineered code often results in visual and atomic classes. We know how painful they are because we discussed them thoroughly in semantics. Mixins may also be a problem which we'll discuss shortly.

Whilst we often try to abstract CSS too much too soon, there are obviously going to be times when reuse makes sense. The question must be answered, *how can we reuse a style?*

How can we reuse a style?

If we want to reuse a style, one option would be to comma-delimit selectors inside a well-named file, which if you're into SASS is exactly what `@extend` does. For example, if multiple elements need red text, we could do this:

```
.someThing,
.anotherThing {
  color: red;
```

```
}
```

This approach should be used for convenience, not for performance. (If the abstraction only has one rule, we're simply exchanging one line of code for another.)

If a selector deviates from the rules inside the abstraction, it should be removed from the list. Otherwise it could regress the other selectors and cause override issues.

It's important to note that this is one of several techniques at our disposal. When a *thing* is well understood we can make use of other techniques, which we'll discuss in Modules, State and Modifiers.

What about mixins?

Mixins provide the best of both worlds. At least in theory.

Like utility classes, updating a mixin propagates to all instances. If we don't have a handle of what's using the mixin, we increase the risk of regression. Instead of updating a mixin, we can create another, but this causes redundancy.

Also, mixins easily end up with many rules, multiple parameters, and conditionality. This is complicated. Complicated is hard to maintain.

To mitigate this complexity, we can create granular mixins, such as one for red text. At first this seems better. But isn't the declaration of a red mixin, the same as the rule itself i.e. `color: red`?

If we need to update the rule in multiple places, a search and replace might be all that's necessary. Also, when the red *mixin* changes to

orange, its name will need updating anyway.

With all that said, mixins can be very useful. We might, for example, want to apply *clearfix* rules across different elements and only within certain breakpoints. This is something that vanilla CSS can't do elegantly.

As such, mixins are not *bad*, it's just that we should use them judiciously.

What about performance?

We often overthink performance and get obsessed with tiny details. Even if CSS did total more than 100kb, there's little to gain from mindlessly striving for DRYness.

Making CSS small makes HTML big. CSS can always be cached. But HTML often contains dynamic and personalised content—so it can't be cached.

The compression of a single image gives us a better return on investment. And as we've discussed, resolving other forms of redundancy improves maintainability *and* performance.

As you'll see in later chapters, the conventions in this guide, mean CSS class names have repeated prefixes which works especially well with GZip.

Is this violating DRY principles?

Attempting to reuse, for example `float: left`, is akin to trying to reuse variable names in different Javascript objects. It's simply not in violation of DRY.

Final thought

Striving for DRY leads to over-thought and over-engineered code. In doing so we make maintenance harder, not easier. Instead of obsessing over styles, we should focus on reusing tangible modules. Something we'll discuss in upcoming chapters.

NEXT: IDS

CHAPTERS

Written by Adam Silver, an interaction designer from London, UK. Contribute on GitHub.

CHAPTER 4

IDs

Semantically speaking, we should use an ID when there's only one instance of a thing. And we should use a class when there are several.

However, IDs overpower class names by orders of magnitude, which is a problem when we want to override a style.

To demonstrate the problem, let's override the colour of an element from *red* to *blue* using an ID.

Here's the HTML:

```
<div id="module" class="module-override">
```

And the CSS:

```
#module {
  color: red;
}

.module-override {
  color: blue;
}
```

The element will be red even though the override class declares blue. Let's fix this by swapping the ID for a class:

```
<div class="module module-override">
```

And the CSS:

```
.module {
  color: red;
}

.module-override {
  color: blue;
}
```

Now, the element is blue—problem solved.

Whilst using IDs for styling is problematic, we can still use them for other things. For example, we'll most certainly need to use them to connect:

- labels to form fields
- in-page anchors to a hash fragment in the URL
- ARIA attributes to help screen reader users

Final thought

Use IDs whenever you need to enable particular behaviours for browsers and assistive technology. But avoid using them as hooks for styles.

NEXT: CONVENTIONS

CHAPTERS

Written by Adam Silver, an interaction designer from London, UK. Contribute on GitHub.

CHAPTER 5

Conventions

MaintainableCSS has the following convention:

```
.<module>[-<component>][-<state>] {}
```

Square brackets are optional depending on the module in question. Here are some examples:

```
/* Module container */
.searchResults {}

/* Component */
.searchResults-heading {}

/* State */
.searchResults-isLoading {}
```

Notes:

- component and state are both delimited by a dash
- names are written with lowerCamelCase
- selectors are prefixed with the module name

Must I give a class name to every element?

No. You can write `.searchResults p` if you want to. And sometimes you may have to, if for example you're using markdown. But beware that if you nest a module which contains a `p` it will inherit the styles and need overriding.

Why must I prefix the module name?

Good question. Here's some HTML without a prefix:

```
<div class="basket">
  <div class="heading">
```

And the CSS:

```
/* module */
.basket {}

/* heading component of basket module */
.basket .heading {}
```

There are two problems:

1. when viewing HTML, it's hard to differentiate between a module and a component; and
2. the `.basket .heading` component will inherit styles from the `.heading` module which has unintended side effects.

NEXT: MODULES

CHAPTERS

Written by Adam Silver, an interaction designer from London, UK. Contribute on GitHub.

CHAPTER 6

Modules

What's a module?

A module is a distinct, independent unit, that can be combined with other modules to form a more complex structure.

In a living room, we can consider the TV, the sofa and the wall art modules. All coming together to create a useable room.

If we take one of the units away, the others still work. We don't need the TV to be able to sit on the sofa etc.

In a website the header, registration form, shopping basket, article, product list, navigation and homepage promo can all be considered to be modules.

What's a component?

A module is made up of components. Without the components, the module is incomplete or broken.

For example a sofa is made up of the frame, upholstery, legs, cushions and back pillows, all of which are required components to allow the sofa to function as designed.

A logo *module* might consist of copy, an image and a link, each of

which are components. Without the image the logo is broken, without the link the logo is incomplete.

Modules vs components

Sometimes it's hard to decide whether something should be a component or a module. For example, we might have a header containing a logo and a menu. Are these components or modules?

In a recent project it made most sense for the logo to be a component and the menu to be a module of its own. What's a header without logo? And the navigation might be moved below the header.

Nobody understands your requirements as well as you do. Through experience you'll get a feel for it. And if you get it wrong, changing from a component to a module is easy.

That's enough theory. Let's build three different modules together. In doing so, the hope is to cover most of the things we think about when writing CSS.

1. Creating a basket module

We'll simplify this basket for brevity. Each product within the basket will display the product's title with the ability to remove it from the basket.

The basket template might be:

```
<div class="basket">
    <h1 class="basket-title">Your basket</h1>
    <div class="basket-item">
```

```
      <h3 class="basket-productTitle">Product title<
      <form>
        <input type="submit" class="basket-removeBut
      </form>
    </div>
</div>
```

And the CSS would be:

```
.basket {}
.basket-title {}
.basket-item {}
.basket-productTitle {}
.basket-removeButton {}
```

2. Creating an order summary module

Next, we will build an order summary module. This module is shown during checkout and bears some resemblance to the basket. For example, it has a title and it displays a list of products.

It does, however, have a different aesthetic and the products can no longer be removed i.e. no form and no remove button.

The first thing to address is the temptation to reuse the basket template (and CSS). Even though there are similarities, this does not mean they are the same.

If we try to combine them we'll entangle two modules with display logic and CSS overrides. This entangling by definition is complex which in turn is hard to maintain and easily avoidable.

Instead, we should create a new module with the following template:

```
<div class="orderSummary">
  <h2 class="orderSummary-title">Order summary</h2>
  <div class="orderSummary-item">...</div>
  <div class="orderSummary-item">...</div>
</div>
```

And the CSS would be:

```
.orderSummary {}
.orderSummary-title {}
.orderSummary-item {}
```

As counterintuitive as this may seem, duplication is a better prospect. And, this is not really duplication. Duplication is copying the *same* thing. These two modules might look similar but they are not the same.

Keeping things separate, keeps things simple. Simple is the most important aspect of building reliable, scalable and maintainable software.

3. Creating a button module

As our basket module only appears in the basket page, we didn't consider being able to reuse it elsewhere. Also, we didn't address the fact that the remove button is a component of the basket, making it harder to reuse across modules.

Buttons are an example of something that we want to reuse in lots of

places, and potentially *within* different modules. (A button is not particularly useful on its own.)

One option would be to upgrade the button component into a module as follows:

```
<input class="button" type="submit" value="{{text}}"
```

And the CSS would be:

```
.button {}
```

The problem is that buttons often have slightly different positioning, sizing and spacing depending on context. And of course there are media queries to consider.

For example, within one module a button might be floated to the right next to some text. In another it might be centered with some text beneath with some bottom margin.

Ideally, we should iron out these inconsistencies in *design*, before they even make their way into code. But as this is not always possible and for the purposes of example, we'll assume we have to deal with these issues.

And so, because of these differences, it's tricky to abstract the common rules because we don't want to end up in override hell. Or worse that we're afraid to update the abstracted CSS rules.

To avoid these problems, we can use a mixin or comma-delimit the

common rules that aren't affected by their context. For example:

```
.basket-removeButton,
.another-loginButton,
.another-deleteButton {
  background-color: green;
  padding: 10px;
  color: #fff;
}
```

Notice that in this example, we don't specify `float`, `margin` or `width` etc. Those styles are applied to the unique button:

```
.basket-removeButton {
  float: right;
}

.another-deleteButton {
  margin-bottom: 10px;
}
```

This seems sensible as it means we can opt in to these common styles. The opposite, of course being having to override. But there's another, third option.

Imagine a checkout flow whereby each page has a continue button and a link to the previous step. We can reuse this by upgrading it into a module:

```
<div class="checkoutActions">
```

```
  <input class="checkoutActions-continue">
  <a class="checkoutActions-back"></a>
</div>
```

And the CSS would be:

```
.checkoutActions-continue { }

.checkoutActions-back { }
```

In doing this, we abstracted and applied the styles to a well understood `.checkoutActions` module. And we've done this without affecting similar, but not identical buttons.

We haven't discussed having more than one type of button (primary and secondary etc) yet. To do this we can use modifiers, which is addressed later.

Final thought

A module, by definition, is a reusable chunk of HTML and CSS. Before a group of elements can be upgraded into a module, we must first understand what it is and what its different use cases are.

Only then, can we design the right abstraction. And in doing so, we avoid complexity at the same time, which is the source of unmaintainable CSS.

NEXT: STATE

CHAPTERS

Written by Adam Silver, an interaction designer from London, UK. Contribute on GitHub.

CHAPTER 7

State

Quite often, particularly with richer user interfaces, styling needs to be applied in response to an element's change of state. For example, we may have different styles when a module (or component) is:

- `showing` or `hiding`
- `active` or `inactive`
- `disabled` or `enabled`
- `loading` or `loaded`
- `hasProducts` or `hasNoProducts`
- `isEmpty` or `isFull`

To represent state we need an additional class which should be added to the module (or component) element to which it pertains. For example, if our basket module needs a gray background when it's empty, the HTML should be:

```
<div class="basket basket-isEmpty">
```

And the CSS should be:

```
.basket-isEmpty {
```

```
  background-color: #eee;
}
```

The class name is prefixed with the module (or component) because whilst states might be common, associated styles might not. For example, an empty *basket* has a gray background, where as an empty search has an absolutely-positioned image.

What about reusing state?

Sometimes, we may in fact want to reuse state across modules or components. For example, toggling an element's visibility. This is discussed in more detail in the chapter entitled Javascript.

What about ARIA attributes?

Not all visual states can be represented by an ARIA attribute. For example, there's no attribute to represent `hasProducts`. Therefore, we should use them only when necessary and in *addition* to classes.

Also, using an attribute (instead of a class) selector has less support. Whilst developers may consider these browsers old, insecure or irrelevant, we should avoid techniques that unnecessarily exclude users.

What about chaining classes?

We could use a chained selector for state e.g. `.module.isDisabled`. The problem is that this approach has less browser support. We should avoid patterns that unnecessarily exclude users, unless there's a compelling reason to do so.

This also makes it harder to find out if/where the style is used, which

makes maintaining harder.

Final thought

If an element's style needs changing based on its state, we should add an extra class to apply the differences. When necessary, use ARIA attributes for assistive technology, not for styling. In doing so we employ a consistent and inclusive approach to styling.

NEXT: MODIFIERS

Written by Adam Silver, an interaction designer from London, UK. Contribute on GitHub.

CHAPTER 8

Modifiers

Like state, modifiers also override styles. They are useful when modules (or components) have small and well understood differences.

Take an e-commerce site whereby each category has a unique background image in the header. All headers have the same padding, and margin etc. The only difference is the background image.

The boys category would have a modifier as follows:

```
<div class="categoryHeader categoryHeader--boys">
```

And similarly, the girls category would have a *girls* modifier:

```
<div class="categoryHeader categoryHeader--girls">
```

The CSS would be:

```
.categoryHeader {
  padding-top: 50px;
  padding-bottom: 50px;
```

```
  /* etc */
}

.categoryHeader--boys {
  background-image: url(/path/to/boys.jpg);
}

.categoryHeader--girls {
  background-image: url(/path/to/girls.jpg);
}
```

Because the differences are small and well understood, this type of reuse is more maintainable.

We can use the same approach for buttons. Most sites have a primary and secondary button style. If all that changes is one or two styles we can have a modifier for primary and secondary buttons as follows:

```
.button {
  padding: 20px;
  border-radius: 3px;
  /* etc */
}

.button--primary {
  background-color: green;
}

.button--secondary {
  background-color: #eee;
```

```
}
```

Again, this only works because the differences are well contained and well understood.

Final thought

Modifiers are a good way to reuse styles across a well understood element. But, the modifier itself should be a tweak. If it contains a lot of overrides, then modifiers are not the way to go. Instead use a module.

NEXT: VERSIONING

Written by Adam Silver, an interaction designer from London, UK. Contribute on GitHub.

CHAPTER 9

Versioning

We may, for example, want to A/B test two different versions of a module to see which works best. To do this, we need to duplicate the module and give it a unique name. For example, if we want to test two different baskets, the CSS might be as follows:

```
/* existing module (variant A) */
.basket {}

.basket-title {}

/* new version (variant B) */
.basket2 {}

.basket2-title {}
```

This way we can maintain two versions during testing until we settle on the best one. And, once we do, it's easy to discard the redundant module as they are not intertwined. Good code is easy to delete.

NEXT: JAVASCRIPT

CHAPTERS

Written by Adam Silver, an interaction designer from London, UK. Contribute on GitHub.

CHAPTER 10

Javascript

We may want to use Javascript to apply the same behaviour to multiple modules or components.

For example, we may use a `Collapser` constructor that toggles an element's visibility.

There are two approaches we can take, both of which complement the CSS approach we've discussed in previous chapters.

1. Encapsulating state to the module

To do this, we would need to specify a module-specific state class to the constructor as follows:

```
var module1Collapser = new Collapser(element1, {
  cssHideClass: 'moduleA-isHidden'
});

var module2Collapser = new Collapser(element2, {
  cssHideClass: 'moduleB-isHidden'
});
```

Then reuse the CSS styles as follows:

```
.moduleA-isHidden,
.moduleB-isHidden {
  display: none;
}
```

The trade-off is that this list could grow quickly (or use a mixin). And every time we add behavior, we need to update the CSS. A small change, but a change nonetheless. In this case we might consider a global state class.

2. Creating a global state class

If we find ourselves repeating the exact same set of styles for multiple modules, it might be better to use a global state class as follows:

```
.globalState-isHidden {
  display: none;
}
```

This approach does away with the long comma-delimited list. And we no longer need to specify the module class when instantiating. This is because the global class will be referenced from within.

```
var module1Collapser = new Collapser(element1);
var module2Collapser = new Collapser(element2);
```

However, this approach doesn't always make sense. We may have two different modules that *behave* the same, but *look* different, which is something we've discussed in State.

3. The best of both worlds

We could combine the two approaches by defaulting the class to the global state class. And then only when needed we can specify a class during instantiation as shown in the first example above.

Final thought

When we think about state, particularly with our Javascript hat on, we need to consider how this state affects behaviour as well as style. Different components may share the same behaviour, but they may look rather different. After careful consideration, we can choose the right solution to the problem.

NEXT: ORGANISATION

CHAPTERS

Written by Adam Silver, an interaction designer from London, UK. Contribute on GitHub.

CHAPTER 11

Organisation

Good code is easy-to-find and easy-to-find code is well-organised. And so it follows we want our CSS to be well-organised. There are, generally speaking, two approaches to choose from, both of which we'll discuss in this chapter.

1. CSS in a single folder

This approach puts all CSS inside a single folder:

```
path/to/css/
  vendor/
    some3rdParty.css
    someOther3rdParty.css
  yourApp/
    some.css
    global.css
    basket.css
```

Notes

- Third-party CSS files live under `/vendor`.
- The application's CSS lives under `/yourApp` where *yourApp* is the name of your project.
- This approach simplifies deployment because a build script can

easily target a single directory in order to bundle and compress the files.

- This seems to be the most common approach but that doesn't mean it's the best.

2. CSS in a module folder

This approach puts module-specific CSS within a folder of its own:

```
global/
  css/
    resetPerhaps.css
    global.css
    etc.css
basket/
  controllers/
    ...
  templates/
    basket.html
    emptyBasket.html
  partials/
    basketHeader.html
    basketSummary.html
  js/
    ...
  css/
    basket.css
header/
  ...
```

Notes

- We normally orientate ourselves by feature as opposed to technology, making this approach a compelling one.
- Global CSS needs a folder of its own because global styles by their very nature don't belong to a module.
- This approach is more likely to suffer from the *31 CSS file limit problem*, which is explained next.

The 31 CSS file limit problem

Whichever approach you take, be aware of the 31 CSS file limit found in versions of Internet Explorer. Internet Explorer 9, for example, ignores styles stored in the 32nd (or 33rd etc) file.

For production this is fine, because we should bundle our CSS to reduce HTTP requests. But for local development it's better to work with source files to make debugging easier. And it's in legacy browsers where bugs normally arise.

If you have a compilation step for local development—as would be the case when using a CSS preprocessor—you don't need to worry. The preprocessor will bundle the files.

If you don't have a compilation step for local development —because debugging source files is easier this way—then you may want to remedy this with one of two approaches:

1. Add an option to concatenate CSS locally

By doing this you'll be able to mimick production and debug CSS in offending legacy browsers.

2. Use less than 32 CSS files

As you'll probably have more than 31 modules, you can't organise your CSS by module. Instead you'll have to put several modules within the same CSS file.

Final thought

In this chapter we've discussed two ways in which to organise CSS. Whichever approach we take, we should be aware of the 31 CSS file limit problem because it makes debugging CSS much harder in the very browsers that cause most trouble.

NEXT: FAQS

Written by Adam Silver, an interaction designer from London, UK. Contribute on GitHub.

CHAPTER 12

FAQs

If you can't find an answer here, raise an issue on Github.

Can I translate this?

Yes but please cite the original (that's this site) and let me know.

What about inheritance for headings etc?

Ideally our semantic HTML matches the integrity of the visual design. Meaning that we would hope that all `h1`s are identical. In this case we can declare the following CSS:

```
h1 {
  /* etc */
}
```

However, this is rarely the case, in commercial, large-scale websites. In this case we should encapsulate styles to the module in question:

```
.module-heading {
  font-size: ...;
  color: ...;
}
```

CHAPTERS

Written by Adam Silver, an interaction designer from London, UK. Contribute on GitHub.