SYSTEMATIC WEB DESIGN

Kieran Potts

Systematic Web Design
© 2015 Kieran Potts

First published (draft) March 2015
Last updated September 2015

Updates

Download the latest version of this book from
http://www.kieranpotts.com

Typefaces

Merriweather, Merriweather Sans, DejaVu Sans Mono

Contents

Introduction

As the name suggests, Cascading Style Sheets set styles in a cascading fashion. Multiple style rules may be applicable to any one element in the HTML markup. Which rules get applied is determined by cascading down through *every* ruleset in *every* style sheet looking for matches. The more specific rulesets override the more general ones. In addition, some styles once applied to an element will cascade downwards and apply themselves to all of that element's descendants, too.

This inheritance model is intrinsic to CSS and cannot be turned off.

In simple, small-scale user interfaces the style cascade can be hugely beneficial. We can set a property just once and have it trickle through to every relevant part of our design. The result: small file sizes, efficient rendering and highly consistent user interfaces.

But the style cascade can also be a bit of nuisance. The CSS inheritance model tightly couples the presentation of every component of a web page, and that makes it difficult to change stuff. Every change has the potential to inadvertently alter the presentation of other things. We end up frequently resetting bunches of inherited properties that we don't want, which rather negates the benefits of the cascade.

The bigger the project, the more the style cascade is a hindrance than a help. It's why CSS is notoriously difficult to maintain in high-fidelity designs (where there is lots of detail and variation in the presentation of things), in long-running projects with many collaborators, and in projects where the team members have widely differing specialities and abilities or where there is a high staff turnover. In these scenarios if everyone is not on the same page in terms of how to structure and manage CSS, a codebase can rapidly snowball into an unmanageable mess. And that's all it takes to cripple a web startup.

CSS does not scale at all well for today's richly formatted, rapidly iterated websites.

Extended CSS languages – Sass, Less, Stylus, *et al* – make things a bit better. And some of their best ideas – like variables – should get ported to native CSS in time. But extended CSS languages don't truly fix the scalability issue. They just skirt around the problem by encouraging code duplication, deep nesting of selectors and other bad ideas.

What we really need is a proper scoping mechanism, a standardised way to encapsulate styles within fragments of markup. This would allow us to split up complex designs into lots of bite-size modules of HTML, CSS and optionally JavaScript. Each module could be developed and maintained separately of the rest. Team members could more easily work on different things in parallel. And it would be easier to take UI components from previous projects and plug them straight in to new projects, speeding up development cycles.

Modular CSS would be the ultimate scalable CSS.

There is hope. The Shadow DOM – part of the exciting Web Components standard dreamt up at the W3C – will encapsulate styles by default. A new `@scope` rule is at an early proposal stage. And the `scoped` attribute, proposed for the `<style>` element, will limit the application of styles to the part of the DOM tree where the `<style>` element is located.

```
<div>
    <style scoped> h2 { color: red; } </style>
    <h2>This is red</h2>
</div>
<h2>This is NOT red</h2>
```

In addition, soon we'll be able to use a new property called "`all`" to fully reset things back to their default presentation, effectively restarting the inheritance chain.

```
*.reset-me { all: unset; }
```

None of these things are useful right now, because they do not yet work consistently in all mainstream browsers. The Shadow DOM is implemented only in Chrome, Opera and Android 4.4's native browser, and it is an experimental feature in Firefox. The `all` property works only in recent editions of Chrome, Firefox, Opera and Android 4.4.4+. As for scoped CSS, its future is highly uncertain. The `scoped` attribute showed up briefly in Chrome (behind a flag) but subsequently since vanished, and Microsoft says it has "no current plans" to implement scoped styles in Edge. At this time scoped styles work only in Firefox.

No matter. Things are certainly moving in the direction of modularity. In the future we will have standardised methods of scoping presentational styles to particular bits of an HTML document and locking out the inheritance chain from others. This will make it possible to develop web interfaces as a series of stand-alone UI components. And that will make it much easier to manage the codebases of large-scale web projects.

In the meantime we must find a way to do this ourselves. One option would be to polyfill Web Components in browsers that lack native support (http://webcomponents.org/polyfills/). Personally, I'm not keen on the idea of building production-ready web apps on emerging technologies that are still in a state of flux. There is still disagreement among browser vendors over implementation of some aspects of Web Components, such as HTML imports which are effectively on hold due to the overlap with ECMAScript 6 module loading. Furthermore, polyfills are heavy and the correct rendering of baseline content becomes dependent upon JavaScript. Bespoke solutions such as CSS Modules (https://github.com/css-modules) have the same problem.

Right now, the best option is to devise our own system to manage CSS at scale.

Web design methodologies

A web design methodology is a system for applying CSS to HTML. The primary aim is to make it possible to break up big, complex web designs into lots of smaller, simpler modules of HTML and CSS (and optionally JavaScript) that can be developed and maintained independently of one another. But a good web design methodology will achieve much more. It will:

- Make the design process faster.
- Make front-end code easier to read and understand.
- Make it easier for multiple people to collaborate on a design.
- Make for a smoother transition in the handover of a project to a new team.
- Reduce the learning curve for new designers joining a project.
- Help developers understand the relationship between HTML and CSS in a project.
- Make it easier to change things.
- Make it easier to scale a design to accommodate more content and functionality.
- Make it easier to add more variety to the visual presentation.
- Make it easier to reuse things that already exist in a project codebase.
- Make it easier to port UI components from a previous project into a new one.
- Encourage production of well-formed, accessible, standards-compliant code.
- Enforce consistency and limit duplication and bloat.
- Reduce page size and increase page rendering speed.
- Provide ready-made documentation.

– Increase productivity.
– And make web design a more rewarding, enjoyable experience.

Web design methodologies benefit all kinds of web projects, big and small. They benefit freelancers who work solo on small-scale web projects. And they are absolutely essential for large and long-running projects.

A few entrepreneurial types have developed their web design methodologies into formal design systems and published them for everyone's benefit. Nicole Sullivan's Object-Oriented CSS Framework (OOCSS), published in 2009 (https://github.com/stubbornella/oocss/), was the first web design methodology to become widely adopted. It is still hugely influential. Jeremy Clarke's Don't Repeat Yourself principles (DRY CSS), presented at ConFoo in 2012 (https://vimeo.com/38063798), has similar goals to OOCSS, but offers a very different solution. Brad Frost followed up with a widely circulated blog post titled "Atomic Design" in 2013 (http://bradfrostweb.com/blog/post/atomic-web-design), which got us thinking about web interfaces as a hierarchy of progressively more complex objects built up from lots of smaller ones. Other popular web design methodologies include: Block Element Modifier (BEM, http://bem.info), developed by the dev team at Yandex; Jonathan Snook's Scalable and Modular Architecture for CSS (SMACSS, http://smacss.com); and Nicolas Gallagher's SUIT CSS (http://suitcss.github.io), which combines a class naming system with extended CSS syntax.

About this book

This book presents a brand new web design methodology – mine. This is how I write and manage CSS at scale. I call it Systematic Web Design. It is a solution that I have developed over a period of at least three years. It is influenced by many of the ideas and conventions of existing web design methodologies, though the practical implementation is generally simpler. There is less to remember.

An abbreviated version is published online at http://www.systematicwebdesign.com

Systematic Web Design – or just "Systematic" for short – is a pure CSS methodology, though it works equally well whether you are writing Sass or Less or any other superset of CSS. Personally I use only a limited amount of extended CSS syntax, just variables and a few mixins. Most of the other features of extended CSS languages become redundant once you've got a solid web design methodology in place.

Systematic Web Design prescribes strict class naming conventions, which reflect how this design system separates out the various concerns in a user interface into discrete units of code. But that's all it does. Systematic Web Design does not wander into the territory of guidelines, build tools or workflow processes. The coding style you adopt in writing CSS, and how you comment and document your code, is left up to you. Style guides, UI pattern libraries, tools like preprocessors and linters, frameworks like Bootstrap and Skeleton, and formal workflow systems like Scrum and pair programming – all of these things can compliment the practical implementation of a web design methodology like Systematic, but these things are outside of the scope of this book.

The first chapter of this book summarises the software design principles – things like encapsulation, loose coupling and defensive programming – that have influenced the Systematic Web Design methodology. The next four chapters are dedicated to the four component parts of Systematic Web Design: elements, layout, widgets and modifiers. There follows a chapter that summarises Systematic's class naming conventions, and a short closing chapter deals with the practicalities of applying Systematic Web Design in the context of responsive design and CSS preprocessors.

There is no "best" web design methodology. Different approaches work better for different types of projects. What matters more than the specifics of any particular web design methodology is the consistency with which one is implemented within a codebase. I can't encourage you to religiously execute my web design methodology in every project when I don't even do that myself. But this is always my starting point. Systematic Web Design is elementary enough that it can be easily tweaked and extended on a project-by-project basis, moulded to better fit the skill set and creative temperament of the wider team.

Web design methodologies have a long future. Even with standard systems for modular UI design on the horizon, there will always be a need for sober, articulate front-end architecture.

About the author

I am a technology-journalist-turned-web-developer from Bath, England. I specialise in the development of bespoke websites and online applications for technology startups. My email is hello@kieranpotts.com.

This book is free, Creative Commons-licensed, and regularly updated. If you would like to be informed when new editions of this book become available for download, please sign up to my mailing list. Click the newsletter link on my website, http://www.kieranpotts.com, and follow the instructions.

Principles

All web design methodologies are based on similar principles and adopt similar design patterns, though emphasis varies from one method to another. These are the principles that have most influenced Systematic Web Design.

Embrace constraints

To reduce the influence of the style cascade and thereby reduce the frequency with which stuff needs to be reset, some web designers adopt a system of giving everything a class, and then assigning styles to elements via their classes rather than directly via their types. So instead of this ...

```
ul {
    color: hsl(0, 0%, 27%);
    font-size: 1.2rem;
    list-style: square inside;
    margin: 0 0 1em;
}

nav ul {
    color: initial;
    font-size: 1rem;
    list-style: none;
    margin: 0;
}
```

... you'd do this:

```
.list {
    color: hsl(0, 0%, 27%);
    font-size: 1.2rem;
    list-style: square inside;
    margin: 0 0 1em;
}

.nav-list {
    font-size: 1rem;
}
```

There are many good things to be said for this approach. If we use classes as the exclusive contract between HTML and CSS we will benefit from very granular control over the presentation of individual bits of our design. Our CSS selectors will be less dependent upon particular HTML structures, allowing us to change much more HTML before triggering wider CSS refactoring. And we will rarely need to unwind unwanted inherited styles, because element types need be given very few default properties in the first place.

(Some would argue that greater dependence on classes increases the coupling between HTML and CSS. This is wrong. Classes are intended exactly for this purpose. Classes are an abstraction layer linking two separate concerns, the equivalent of hooks in event-driven programming.)

But the class-based approach is far from perfect. We end up with HTML documents that are riddled with classes. And because we're bypassing much of the cascade we don't benefit from the convenience and efficiency of the CSS inheritance model. Plus class-based designs tend to have a lot more rulesets overall, giving browsers more work to do to resolve the final styling of each element.

And how far should we take this approach, really? Is it sensible to apply classes to every paragraph, every list and every image, just to get some default styling in place? Should we go even further and strip default styles from elements like `<a>`, `<strong>` and `<em>`? Of course not. That's utterly impractical.

There is a balance to be found between harnessing the benefits of the cascade by assigning lots of default properties for lots of elements, and locking things down by using classes as the primary API linking HTML to CSS. The sweet spot between these two extreme approaches will vary from project to project. But our default position should be to make the most of the style cascade.

We should always start a new web design project by defining default styles for base typography, hyperlinks, tables, input controls, buttons, and indeed for every visible element that will be needed to compose the design. These defaults will trickle through to every context in which each element type is used, and that's okay. We must accept that we will need to reset an element's default styles sometimes, when it is used in special contexts. If we find ourselves resetting lots of stuff, we can iterate – in a restrained and considered way – towards reducing the number of properties applied via naked type selectors and increasing our use of classes.

Embracing the cascade from the outset, rather than trying desperately to work around it, helps to keep our creative impulses in check, too. It encourages us to build visual style guides around the limited number of native HTML elements rather than the infinite possibilities afforded by classes. The natural constraints of the cascade make it difficult to create anything except supremely consistent designs that are quickly rendered in the browser. That's a good place to start, at least.

Encapsulation

The underlying cause of the ills of the CSS inheritance model is that it makes everything global. When the browser is presented with a new element to render, it will traverse *every* selector in *every* style sheet looking for matches. Thus every ruleset has the potential to be applicable to everything on the web page. This is what makes CSS so inherently leaky.

Until we get a standard scoping mechanism for CSS, we must devise our own scoping mechanism. The answer is to use classes as namespaces. Namespacing helps to stop styles from leaking in and out of distinct parts of an interface, like sidebars and footers.

Consider the following example.

```
.sidebar a {
    background: hsl(195, 5%, 80%);
    color: hsl(0, 0%, 100%);
    padding: 0.25em 0.5em;
    text-decoration: none;
}
```

Hyperlinks nested in our website's sidebar will of course inherit the default properties assigned for all anchor (`<a>`) elements. The above ruleset then modifies and extends those defaults, but only for hyperlinks that are located within the sidebar. The containing `"sidebar"` class acts as a namespace, encapsulating certain presentations within a particular section of the document.

But what would happen if we added to the sidebar, say, a little calendar widget? The above ruleset would apply itself to links within the calendar's markup, too. That is almost certainly not what the designer had intended.

One of the key principles of class-based encapsulation is to be very precise and intentional about what we select for styling. One possible way to achieve this is to use class-based namespacing in conjunction with descendant selectors to zero-in on precisely the things that we want to target.

```
.sidebar nav > ul a {}

.sidebar .calendar table a {}
```

But new problems arise with such overly long descendant selectors. The first problem is that rulesets become dependent upon there being very specific HTML structures in place. In the example above the presentation of the navigation links depends on them being nested inside an unordered list, which must be the immediate descendant of a `<nav>` element, which in turn must be nested in something with class `"sidebar"`. This is a simplified example, but the point is made: if the HTML changes, the design breaks.

Also, long descendant selectors like these have a high specificity. Imagine if later we wanted to vary the presentation of certain `<a>` instances that appear within the calendar widget. We'd have to create new selectors that have an even higher specificity to compete with these existing selectors. High-specificity selectors make it difficult to modify and extend things because the only way to do so is to wage specificity wars, selector against selector!

There is one other good reason to avoid complex combinator selectors. Consider the following example.

```
.sidebar section:first-of-type h2 + p {}
```

Modern browsers evaluate selectors like this very quickly indeed. Performance is no longer a big issue when it comes to CSS selector formats. But complex combinator selectors are still pretty tricky for the silly old human brain to evaluate. I bet it will take you a few moments to figure out exactly what things this ruleset affects. (Answer: the first paragraph placed immediately after every `<h2>` element in the first `<section>` of the sidebar.)

For our own sanity as much as anything else, we need a looser coupling of CSS to HTML.

Loose coupling

The mark of a maintainable software program is when developers can easily and confidently change things in one part of the codebase without inadvertently breaking things in other parts. In front-end web engineering this means being able to alter the source order of an HTML document without triggering major CSS refactoring.

Here's our selector again for hyperlinks in the sidebar menu.

```
.sidebar nav > ul a {}
```

And here's the selector rewritten to be more loosely coupled to the HTML without compromising on the level of encapsulation. It uses fewer descendant selectors, but compensates with more targeted encapsulation classes.

```
ul.sidebar-menu a {}
```

We could go further still and apply the `"sidebar-menu"` class directly to each nested `<a>` element instead of their shared parent, the unordered list element. We would do away with the descendant selector altogether and the presentation of the sidebar menu links would still be adequately encapsulated – more so, in fact.

```
a.sidebar-menu {}
```

Applying classes directly to the things that we want to look different from their default presentation has numerous benefits. It gives us a bit more flexibility to move things about and to add new stuff. In this case we could easily add new links to the sidebar menu that are styled differently from the existing links. The designer's intention is clearer, too. The classes applied to elements describe exactly what those elements should look like, making it easier to tell if the rendered effect is intentional or a mistake.

The only downside is that we have a lot more classes littered about our markup. In this case the `"sidebar-menu"` class will be duplicated on every link in the sidebar menu. As with everything in CSS, nothing is perfect. There is a balance to be found. Our preference should be to assign classes directly to the elements that we want to differentiate from their default presentation. A flat hierarchy of simple type and class selectors will make our web designs easier to maintain and scale in the long-term.

```
h1 {}
h2 {}
h3 {}

p {}
p.large {}
p.x-large {}

ul {}
ul.no-bullets {}
```

But sometimes it will be perfectly sensible to push encapsulation classes up one or two levels in the DOM tree, as long as doing so still provides adequate encapsulation and reasonable decoupling of HTML from CSS. Attaching encapsulation classes to ancestors is certainly acceptable for UI components made from lists and tables and anything else where the HTML structure is unlikely to ever change.

```
table.account-balance {}
table.account-balance thead {}
table.account-balance th {}
table.account-balance td {}
table.account-balance td:first-child {}
```

Defensive programming

In many of the examples above I have qualified class selectors with type selectors. This still leaves some coupling between HTML and CSS. For even looser coupling we might choose not to specify the types of elements to which our classes are meant to be applied. Below are the selectors again for the `"account-balance"` table, with the `<table>` type qualifier now removed. This gives us a bit more flexibility as to where in the DOM we place the `"account-balance"` class. We could keep it attached to the `<table>` element or we could move it up to its container element, or any other ancestor all the way up to the document body.

```
.account-balance {}
.account-balance thead {}
.account-balance th {}
.account-balance td {}
.account-balance td:first-child {}
```

I come from a programming background, rather than a design background, and I am a big advocate of defensive programming. This is the principle that we should design components of programs for explicit purposes and to report errors when components are misused so that developers are forced to fix the issue before it snowballs into an even nastier bug. For example we might throw exceptions when a function receives an input argument of a type that it is not expecting.

I have adopted the same mentality in my approach to CSS. I prefer to be explicit about the contexts in which my classes are designed to be used. For example I might design the class `"dropcap"` for use on paragraphs of text. So I am clear about that.

```
p.dropcap {}
```

Now if the `"dropcap"` class is accidentally applied by another designer to another type of element, no harm done. The `p.dropcap` ruleset will not get selected. But if I defined the `"dropcap"` class as a global there is a risk that other developers might misuse the class, potentially rendering things in a way that I had not intended.

```
.dropcap {}
```

There are exceptions. Some classes will be designed to be applied to just about anything. In these rare cases we can be explicit that the classes are indeed intended to be used globally by qualifying them with the universal selector (*).

```
*.clearfix {}
```

Classes should not be qualified against element types that are interchangeable with others. Classes should certainly never be qualified against the two non-semantic elements, `<div>` and `<span>`. And I would include the semantic sectioning elements – `<main>`, `<header>`, `<footer>`,

`<article>`, `<aside>`, `<address>`, `<nav>`, and `<section>` – in my list of banned qualifying types, too. It's pretty handy being able to swap around these elements in the markup – for example replacing an `<aside>` element with a standard `<div>` – without needing also to refactor the CSS.

Predictability

Classes help us to scope styles to particular bits of markup. But there is still a real possibility of leakage if we choose class names that are too generic. Consider the following example.

```
<p class="tagline">Tagline</p>
<h1>Headline</h1>
<p class="meta">Author, Date</p>
```

Here are the selectors we'd write to target these elements.

```
h1 {}
p.tagline {}
p.meta {}
```

Now, in a large project it is entirely conceivable that the classes `"tagline"` and `"meta"` could turn up elsewhere. And that may result in styles leaking out from one thing and into another unrelated thing. That's fine if we had intended `"tagline"` and `"meta"` to be global classes, usable in different contexts. But such inclusive terminology should not be used where styles are meant to be locked down to particular parts of an interface.

A robust, stable, scalable CSS codebase is one where all of the classes behave predictably. Just by reading the name of a class among some markup, we should be able to confidently predict what sort of presentational effects the class discharges and the sphere of the class's influence. We should understand exactly what other parts of the application may depend upon the class. And we should understand what would be the consequences of removing the class from the document.

The bigger the project, and the more complex the visual design, the more important it is that classes behave predictably. The way we achieve this is through consistent execution of a dependable class naming methodology.

Separation of concerns

The problem with classes is that they are made to do so much stuff. Some classes set the position of things. Some modify the default presentation of individual elements. Some encapsulate the styles of larger UI components such as navigation bars and modal popups. Some act as targets for DOM queries in JavaScript. Some get dynamically added to documents for the purpose of feature detection (*à la* Modernizr). Hell, classes have even been used as meta data (think microformats).

Consider the following snippet of markup.

```
<div class="header">
```

What does the class `"header"` do here, exactly? Does it position the `<div>` element on the page? Does it style the `<div>`, giving it a background, borders, margins and padding? Does it set the presentation of content that is nested within the `<div>`? Does it act as a target for DOM queries in JavaScript? Of course, we don't know. To find out we'd have to research the class by looking it up in the style sheets or using the browser console.

In a more ideal world we should know the answers to these questions from the name of the class alone. Just by reading the source of an HTML document we should know:

- What each class in the markup does.
- Where a class has come from.
- Where else a class might be used.
- What else a class might be related to.
- What will be the consequences of removing a class.

The problem can be remedied by using different naming conventions to distinguish between classes that fulfil different roles.

Class naming conventions are the backbone of modern web design methodologies.

Clarity is key here. We don't want an overly complex naming convention that demands a steep learning curve. Below are the formats for a class naming convention that is widely used today (Block Element Modifier, or BEM). It's a superb system for categorising the roles of classes and for modularising front-end interfaces, but it produces verbose and somewhat ugly code that is likely to mystify anyone who has no experience of BEM.

```
.component-name {}
.component-name--modifier-name {}
.component-name__sub-object {}
.component-name__sub-object--modifier-name {}
```

Ideally, we want a class naming convention that is as robust and expressive as BEM, but simpler and more intuitive, too.

Single responsibility principle

It is easier to give a class a meaningful name if it does only one very specific thing. Specialist classes are the most predictable of classes, because their names tend to be self-explanatory. I think we can be pretty confident what a class called `"box-shadow"` will do. It is less clear what a class called `"box"` might do. It might apply a box shadow, but equally it could apply borders, padding, backgrounds and maybe even alter the presentation of nested typographic elements and controls. It might do all of these things and more at once.

Classes that are focused on one particular concern tend also to be more useful. We can often reuse specialist classes in lots of different places. Consider the following example.

```
.alert {
    background: hsl(0, 100%, 50%);
    color: hsl(0, 0%, 100%);
    font-size: 1.6rem;
    font-weight: 700;
    left: 0;
    padding: 0.5em 2em;
    right: 0;
    text-transform: uppercase;
    top: 0;
}
```

This ruleset styles and positions a warning message at the top of the screen. Harmless enough. But suppose that down the road we wanted to use the same component but in a different location. Alas we can't reuse our existing `"alert"` class because it has been designed for placement in one specific corner of the layout. The problem is we're trying to make this class do too much. It sets look-and-feel as well as layout and position. So better to split apart these concerns into separate components.

```
.layout-alerts {
    left: 0;
    position: absolute;
    right: 0;
    top: 0;
}

.alert-box {
    background: hsl(0, 100%, 50%);
    color: hsl(0, 0%, 100%);
    font-size: 1.6rem;
    font-weight: bold;
    padding: 0.5em 2em;
    text-transform: uppercase;
}
```

What we've got now is a class called `"layout-alerts"` that positions a container element but does not style any content. And we've got an `"alert-box"` class that *does* style content and wraps it up in a box. By using the two classes together we can compose a presentation that matches that achieved by our original `"alert"` class.

```
<div class="layout-alerts">
    <div class="alert-box">
        ...
    </div>
</div>
```

The point is we can now reuse the "`alert-box`" component in other places. Here it is in the website's sidebar:

```
<div class="layout-sidebar">
    <div class="alert-box">
        ...
    </div>
</div>
```

Composition

We could go further. We could delegate responsibility for other aspects of the presentation to even more classes. Let's create a class called "`warning`" that sets the font colour and background of the alert box. And let's create a class called "`shout`" that transforms text to upper case and renders it in big, bold type.

```
*.warning {
    background: hsl(0, 100%, 50%);
    color: hsl(0, 0%, 100%);
}

*.shout {
    font-size: 1.6rem;
    font-weight: bold;
    text-transform: uppercase;
}
```

Now our original "`alert-box`" class has very little left to do.

```
.alert-box { padding: 0.5em 2em; }
```

The final presentation is composed from multiple classes.

```
<div class="layout-alerts">
    <div class="alert-box warning shout">
        ...
    </div>
</div>
```

This is so much more flexible. If later we wanted to vary the presentation of a single alert box instance, we'd just make new classes. We might want to give an alert box some rounded corners, say.

```
*.rounded-corners { border-radius: 5px; }
```

Or what about a box shadow?

```
*.box-shadow { box-shadow: 7px 7px 5px 0 hsl(0, 0%, 20%); }
```

And here's a class called `"success"` that replaces `"warning"`, changing the background colour of an alert box.

```
*.success {
    background: hsl(120, 100%, 25%);
    color: hsl(0, 0%, 100%);
}
```

Here's the final markup for one particular `"alert-box"` instance:

```
<div class="alert-box success shout rounded-corners box-shadow">
```

Breaking down designs to a very granular level like this allows us to compose lots of different designs from the same base components. We can create a multiplicity of alert box designs. And we can reuse many of the same classes to compose entirely unrelated UI components. Here are some of our newly-forged specialist classes being reused to compose the presentation of a modal popup:

```
<div class="popup success rounded-corners box-shadow">
```

Imagine if the `"rounded-corners"` class also set borders, padding and margins. It would be far less useful. The more specialised the class, the more frequently it can be recycled in the composition of different bits of a UI.

The open/closed principle

Composing complex designs from lots of small, specialised components speeds up the web design process because less CSS needs to be written and maintained. It also makes for a more stable codebase. Specialist classes tend not to need updating as much as comprehensive classes.

Consider the base class we've got now for our alert boxes. It doesn't do much. It just sets some padding. It we wanted to change any aspect of the presentation of an alert box, we'd just extend this base presentation by adding new classes into the mix. We wouldn't need to modify the `"alert-box"` class itself.

```
.alert-box { padding: 0.5em 2em; }
```

And that's very handy. Once a class is forged and imprinted in our UI pattern library, we don't want to have to go back and change it. Doing so may have implications for lots of things that already use that class. And in mature projects that could create a lot more work for sleep-deprived web designers like us.

As a programmer would say, classes should be open for extension but closed for modification.

Don't repeat yourself

The more that we break down a design into small, abstracted, portable components, the less frequently we will need to repeat declarations of the same CSS properties and values. Instead we just reuse the classes that set those properties and values.

But that doesn't help us when we establish the default presentation of individual element types. Consider the following example.

```
h2 {
    font-size: 2rem;
    line-height: 1.2;
    margin-top: 2em;
    margin-bottom: 1em;
}

h3 {
    font-size: 1.4rem;
    line-height: 1.2;
    margin-top: 2em;
    margin-bottom: 1em;
}

p {
    font-size: 1.4rem;
    line-height: 1.2;
    margin-bottom: 1em;
}
```

Writing one selector per element like this is the easiest way to configure the default presentation of element types. But there are lots of duplicated properties in this example, aren't there? All three share the same `line-height` and `margin-bottom` values; the `<h2>` and `<h3>` elements share the same `margin-top` value; and the `<h3>` and `<p>` elements share the same `font-size`.

To eliminate this duplication, we have two options. We could delegate responsibility for recurring default properties to classes. Or we might reorganise type selectors around properties instead of elements, like this:

```
h2          { font-size: 2rem; }
h3, p       { font-size: 1.4rem; }

h2, h3, p   { line-height: 1.2; }
```

```
h2, h3     { margin-top: 2em; }
h2, h3, p  { margin-bottom: 1em; }
```

In this example the number of non white space characters has reduced by 40%. In some of my own projects I have achieved savings of nearly 80%, just by reorganising type selectors around properties rather than elements.

Although we should work hard to maintain DRY ("Don't Repeat Yourself") style sheets, we should not be afraid to allow things to get a little WET ("Write Everything Twice") here and there. If two unrelated UI components happen to have similar visual characteristics, that is purely circumstantial and it is not necessary that the duplicated properties be declared via the same selector. There's nothing wrong with a little bit of repetition if it's the simplest solution.

* * *

Scalable CSS is a structured codebase where the leaky style cascade is controlled. That is done by composing user interfaces from lots of small, autonomous, highly specialised modules of design and dynamic behaviour, each module encapsulated within clear and meaningful class names. These are the principles behind the Systematic Web Design methodology, which is described in the next four chapters of this book.

Elements

The first stage of the Systematic Web Design methdology is to define the default presentation for all of the base HTML elements that you will need to encapsulate content and to render interactive controls. Things that will probably need your attention include:

- Headings, paragraphs, lists and hyperlinks.
- Data tables.
- Forms, fieldsets and legends.
- Input controls and buttons.

Most of the CSS at this stage will be made up of plain old type selectors.

```
h2 {
    font-size: 2rem;
    line-height: 1.2;
    margin-top: 2em;
    margin-bottom: 1em;
}

h3 {
    font-size: 1.4rem;
    line-height: 1.2;
    margin-top: 2em;
    margin-bottom: 1em;
}

p {
    font-size: 1.4rem;
    line-height: 1.2;
    margin-bottom: 1em;
}
```

Though it is easier to deal with the presentation of one element at a time, you might consider reorganising your selectors around properties rather than elements, to reduce duplication of the same properties and values.

```
h2          { font-size: 2rem; }
h3, p       { font-size: 1.4rem; }

h2, h3, p   { line-height: 1.2; }

h2, h3      { margin-top: 2em; }
h2, h3, p   { margin-bottom: 1em; }
```

For efficiency some inheritable typographic styles may be set on the `<body>` element and allowed to cascade from there.

```
body {
    color: hsl(0, 0%, 27%);
    font-family: Roboto, Arial, Helvetica, sans-serif;
    font-size: 62.5%;
}
```

You will probably want to reset some of the default properties set by the browser, to make a more consistent cross-browser base from which to build out your design.

```
* {
    margin: 0;
    padding: 0;
}

html   { font-size: 62.5%; }
html * { font-size: 100%; }

body * { font-family: inherit; }
```

As well as element types, you might want to target pseudo-classes, pseudo-elements and element attributes, too.

```
a:hover { opacity: 1; }
q::before { content: ' «'; }
button[disabled] { opacity: 0.5; }
```

But descendant selectors, child selectors and sibling selectors should be used more sparingly. The default presentation of individual elements should not be dependent upon those elements being used in any particular context.

```
li a {}
li > a {}
h2 + h3 {}
h2 ~ p {}
```

Initially, prefer to promote consistency by fleshing out an exhaustive suite of default styles for individual HTML elements. Later, when you make your modifiers and widgets, if you find yourself resetting lots of inherited properties, consider iterating towards a lower baseline of styling, reducing the number of properties applied directly to elements via their type selectors and increasing your reliance on classes to build up more intricate designs.

There is something to be said for having a very minimalist baseline design. I have an elements style sheet that I use as the "reset" for every web project. This baseline design rarely changes from one project to the other. Because every project is built using the same web design methodology and on the same design foundation, I can easily port UI components – things like tabbed interfaces and modal popups and other such "widgets" – from one project to another.

Layout

The second stage of the Systematic Web Design methodology is to give your HTML documents a structure. Use a series of sectioning elements – `<main>`, `<header>`, `<footer>`, `<article>`, `<aside>`, `<address>`, `<nav>` and `<section>`, and also generic `<div>` elements – to establish sections in which content will be later placed. Give each section a unique class name, written in full upper case with words delimited by underscores.

```
<body>
    <div class="CONTAINER">
        <header class="BANNER"></header>
        <nav class="NAVIGATION_PRIMARY"></nav>
        <nav class="NAVIGATION_SECONDARY"></nav>
        <main class="MAIN"></main>
        <aside class="ADVERTS"></aside>
        <footer class="FOOTER">
            <nav class="SITEMAP"></nav>
            <div class="LEGAL"></div>
        </footer>
    </div>
</body>
```

In the CSS, use class selectors to target each section. Do not qualify the class selectors with types. That will give you the freedom to swap around sectioning elements in the markup – for example, switching an `<aside>` for a generic `<div>`, so changing the semantics of the document – without needing to refactor your style sheets, too.

```
.CONTAINER {}
.BANNER {}
.NAVIGATION_PRIMARY {}
.NAVIGATION_SECONDARY {}
.MAIN {}
.ADVERTS {}
.FOOTER {}
.SIDEBAR {}
.LEGAL {}
```

In mobile-optimised views, document sections will be stacked vertically, one on top of the other. But in larger viewports sections may be positioned to create a layout.

The layout rulesets are concerned exclusively with establishing a basic visual structure in which to place content (which will be styled separately). Layout sections may be positioned and they may be given backgrounds, borders, padding and margins. But layout classes must never set any typographic styles, or alter the presentation of form controls, or try to influence the presentation of anything that may be placed within any section of the layout. The aim is to have the freedom to move content around in the layout without needing to make changes to the CSS.

To this end, do not set any inheritable properties via layout classes. Apply only non-inheritable properties to layout sections. These will not cascade down and affect nested content. Non-inheritable properties include:

- `display`
- `float`
- `position`
- `top`, `bottom`, `left`, `right`
- `width`, `height`, `max-width`, etc.
- `background`, `background-color`, `background-image`, etc.
- `border`, `border-color`, etc.
- `margin`, `margin-left`, etc.
- `padding`, `padding-left`, etc.
- `z-index`

You might not want every page of a website to follow the exact same layout. So you can modify and extend the default layout for different pages. Where a layout differs from the default, give the root `<html>` element a unique class name, using the same naming convention as for layout sections. Example:

```
<html class="SEARCH">
```

Now you can use the `"SEARCH"` class to encapsulate aspects of the layout that are specific to your website's search page. You might make adjustments to sections that already exist in the default layout. And you might introduce entirely new layout sections that exist only on the search page. In the following example there is one selector for modifying and extending the default presentation of the global "NAVIGATION_PRIMARY" section, and there are two selectors for new sections that exist only on the search page: "SIDEBAR" and "PAGINATION".

```
.CONTAINER {}
.BANNER {}
.NAVIGATION_PRIMARY {}
.NAVIGATION_SECONDARY {}
.MAIN {}
.ADVERTS {}
.FOOTER {}
.SITEMAP {}
.LEGAL {}

body.SEARCH .NAVIGATION_PRIMARY {}
html.SEARCH .SIDEBAR {}
body.SEARCH .PAGINATION {}
```

Notice that the "SEARCH" class is qualified with a type selector. This just makes it clearer that it is the encapsulating class for a whole template and not just for one section of the layout.

In designs with lots of layout variations you might consider prefixing global layout sections with "GLOBAL_", to tell them apart from sections that exist only in certain templates.

```
.GLOBAL_CONTAINER {}
.GLOBAL_BANNER {}
.GLOBAL_NAVIGATION_PRIMARY {}
.GLOBAL_NAVIGATION_SECONDARY {}
.GLOBAL_MAIN {}
.GLOBAL_ADVERTS {}
.GLOBAL_FOOTER {}
.GLOBAL_SITEMAP {}
.GLOBAL_LEGAL {}

html.SEARCH .GLOBAL_NAVIGATION_PRIMARY {}
html.SEARCH .SIDEBAR {}
html.SEARCH .PAGINATION {}
```

Alternatively, you might use a grid system to create the various sections in which content will be laid out.

```
<div class="GRID">
    <header class="GRID_ROW">
        <div class="GRID_COLUMN_ONE_OF_TWO">
            ...
        </div>
        <div class="GRID_COLUMN_ONE_OF_TWO">
            ...
         </div>
    </header>
    <main class="GRID_ROW">
        <div class="GRID_COLUMN_TWO_OF_FOUR">
            ...
        </div>
        <div class="GRID_COLUMN_ONE_OF_FOUR">
            ...
        </div>
        <div class="GRID_COLUMN_ONE_OF_FOUR">
            ...
        </div>
    </main>
    <footer class="GRID_ROW">
        <div class="GRID_COLUMN_ONE_OF_ONE">
            ...
        </div>
    </footer>
</div>
```

Personally, I find traditional layouts much more flexible. Because each section is given a unique name, different sections can collapse to mobile-friendly layout at different breakpoints. This is difficult to achieve – though not impossible – with generic grid systems. I *do* use grid systems but not for primary layout. Instead I use grids to set out content *within* an existing layout section. I will cover this technique in the next chapter.

Widgets

HTML's extensive suite of native element types still does not provide us with a large enough vocabulary from which we can design rich, modern web interfaces. We often need to group together multiple HTML elements to achieve a particular effect. We might combine an `<input>` with a `<button>` to create a search box, or repurpose the `<ul>` element to make a navigation menu.

We could just plonk these things down directly inside the sections of the layout where we want them to be.

```
<nav class="NAVIGATION">
    <ul>
        <li><a href="./">Home</a></li>
        <li><a href="about.html">About</a></li>
        <li><a href="learn/">Learn</a></li>
        <li><a href="extend/">Extend</a></li>
        <li><a href="share/">Share</a></li>
    </ul>
    <form action="search.html" method="get">
        <label for="input-search">Search</label>
        <input name="q" type="search" id="input-search" />
        <button type="submit">Search</button>
    </form>
</nav>
```

We would then alter the presentation of individual elements when they appear within a particular layout section by using the name of that section as a target for our CSS selectors.

```
.NAVIGATION ul {}
.NAVIGATION li {}
.NAVIGATION a {}
.NAVIGATION form {}
.NAVIGATION label {}
.NAVIGATION input {}
.NAVIGATION button {}
```

The big disadvantage of this approach is that the presentation of content and interactive controls – in this case, a navigation menu and search box – is tightly coupled to its location within the layout. If we move the content, its presentation breaks. And we can't easily reuse existing markup elsewhere. More work for us.

A better approach is to identify recurring patterns of content in our designs – let's call them "widgets" – and to bundle them up into independent modules of markup and styling (and maybe scripting) that can be freely moved around and reused in different contexts.

A widget is any reusable user interface component. It could be something simple like a search box or a navigation menu. Or it could be something more complex and dynamic like a modal popup, a photo slideshow or a real-time user comment feed. Some widgets may be used on every page of a website, while others may be rolled out only occasionally.

So the next stage of the Systematic Web Design methodology is to design all of the widgets that you will need to render your content and interactive controls. Creating widgets is easy. Simply encapsulate the markup for each widget in a unique class. Widgets use the `"CamelCase"` naming convention, to distinguish them from layout sections (`"UPPER_CASE"`) and modifier classes (`"lower-case"`).

Below, the content for our navigation menu is now encapsulated in a new `<div>` which is given the class `"NavBar"`. And the markup for our search box is now encapsulated in another `<div>` which is given the class `"SearchBox"`.

```
<div class="NavBar">
    <ul>
        <li><a href="./">Home</a></li>
        <li><a href="about.html">About</a></li>
        <li><a href="learn/">Learn</a></li>
        <li><a href="extend/">Extend</a></li>
        <li><a href="share/">Share</a></li>
    </ul>
</div>

<div class="SearchBox">
    <form action="search.html" method="get">
        <label for="input-search">Search</label>
        <input name="q" type="search" id="input-search" />
        <button type="submit">Search</button>
    </form>
</div>
```

Here's the full and final markup, with the NavBar and SearchBox widgets nested inside the original `"NAVIGATION"` layout section.

```
<nav class="NAVIGATION">
    <div class="NavBar">
        <ul>
            <li><a href="./">Home</a></li>
            <li><a href="about.html">About</a></li>
            <li><a href="learn/">Learn</a></li>
            <li><a href="extend/">Extend</a></li>
            <li><a href="share/">Share</a></li>
        </ul>
    </div>
    <div class="SearchBox">
        <form action="search.html" method="get">
            <label for="input-search">Search</label>
            <input name="q" type="search" id="input-search" />
            <button type="submit">Search</button>
        </form>
    </div>
</nav>
```

Each widget's encapsulating class acts as a sort of namespace for presentational styles that are unique to the widget. We no longer need to use the name of a parent layout section as a namespace.

```
.NavBar {}
.NavBar ul {}
.NavBar li {}
.NavBar a {}
.NavBar a:hover {}

.SearchBox {}
.SearchBox form {}
.SearchBox label {}
.SearchBox input {}
.SearchBox button {}
```

Now, the NavBar and SearchBox widgets will look the same no matter where we put them. Their presentation is not dependent upon their markup being located anywhere particular. We can move them about in the layout. We can duplicate them. And we can do these things without touching the CSS.

Widgets should be designed to fit 100% of the width of whatever container we put them in. If you want to float a widget to the left or right side of its container, or if you want to apply fixed or absolute positioning to a widget, better to wrap the widget in a new layout section and position that instead. Keep these concerns – structure and content – separate.

Avoid using semantic sectioning elements – `<main>`, `<header>`, `<footer>`, `<article>`, `<aside>`, `<address>`, `<nav>` and `<section>` – either within or to encapsulate widgets. These elements should be reserved for layout sections. Widgets may be contained in generic divider (`<div>`) elements or any other appropriate block-level element such as a `<ul>`, `<table>` or even a `<blockquote>`.

```
<table class="MiniCalendar">
    <thead>
        <tr class="nav">
            <th class="prev"><a href="?m=02&y=2015">Prev</a></th>
            <th class="this" colspan="5">January 2015</th>
            <th class="next"><a href="?m=02&y=2015">Next</a></th>
        </tr>
        <tr class="days">
            <th>Mon</th>
            <th>Tue</th>
            <th>Wed</th>
            <th>Thu</th>
            <th>Fri</th>
            <th>Sat</th>
            <th>Sun</th>
        </tr>
    </thead>
    <tbody>
        ...
    </tbody>
</table>
```

In practice it will not always be possible to make every widget entirely location agnostic. Some widgets may be positioned in a layout section that has a dark background, in which case the colours of the widget will need to be adjusted accordingly. And responsive breakpoints can be an issue. Breakpoints can be set only according to the size of the viewport or display, and not according to the size of any particular element in the document. Thus the breakpoints that we set for widgets will be dependent upon those widgets being located in a container of a certain size. These are not problems for global widgets, which appear in the same spot on every page: the website's logo, navigation bar, footer links, etc. But these are problems that are worth trying to avoid in widgets that are meant to be reusable in lots of different contexts. In particular, try to avoid relying on media queries to adjust the layout of things *within* a widget. Doing so increases the coupling between widgets and layouts, as the widgets are required to be located in certain sections of the layout.

We could use a combination of JavaScript and CSS to create "element queries" that allow us to adjust the presentation of individual DOM elements based on their rendered size. The Boston-based Filament Group has come up with a simple, lightweight and standards-compliant polyfill, published at the URL below. Use this only as a last resort. Ideally, achieving a correct static presentation should not be dependent upon JavaScript being available. https://github.com/filamentgroup/elementary

Widgets work best when they are quite abstract. A `"ComparisonTable"` widget would be preferable to a `"PriceComparison"` widget, since the former could be reused to compare things besides prices. The more abstract the widget, the greater the opportunity for reusing it for different things.

It is recommended that all content be encapsulated in a widget of some sort. In theory, content can be marked up as a mix of widgets and stand-alone elements. Consider the following example. There is one widget, `"ArticleHeadline"`, which encapsulates an `<h1>` and `<p>` that represent the main headline and trailer text for an article. But the main copy and opening image of the article is placed directly inside the `"MAIN"` layout section. This content will be styled exclusively with the default properties set for the `<img>` and `<p>` element types. (Actually, the presentation of the `<img>` element is extended by application of a global modifier, `"float-right"`. Modifiers are the subject of the next chapter.)

```
<main class="MAIN">
    <header class="HEADLINE">
        <div class="ArticleHeadline">
            <h1>Headline...</h1>
            <p>Strapline...</p>
        </div>
    </header>
    <img src="picture.png" alt="" class="float-right" />
    <p>Lorem ipsum...</p>
</main>
```

This can work well, but in general I find it easier if all visible content and every interactive control is encapsulated in a dedicated widget. It means that when I want to change the design of something I can work exclusively with widgets; I do not have to change the default presentation of individual HTML elements, which will likely have wider consequences. Here's the same content as before but now the main body of the article is encapsulated in a new widget called `"ArticleBody"`:

```
<main class="MAIN">
    <header class="HEADLINE">
        <div class="ArticleHeadline">
            <h1>Headline...</h1>
            <p>Strapline...</p>
        </div>
    </header>
    <div class="ArticleBody">
        <img src="picture.png" alt="" class="float-right" />
        <p>Lorem ipsum...</p>
    </div>
</main>
```

Widgets will, of course, inherit the default properties of the element types that are included in their markup. Those default styles are then changed and extended on a widget-by-widget basis. Sometimes you might want to reset some inherited properties. For example you might want to change the default `color` and `text-decoration` values for `<a>` elements when they are used in the NavBar widget. A few little overrides like this are perfectly acceptable. But if you catch yourself doing lots and lots of resets, consider applying fewer default properties to elements via non-qualified type selectors, so that less stuff is inherited by widgets.

Web interfaces built using the Systematic Web Design methodology will be composed from lots and lots of widgets. You might even have widgets nested inside other widgets. Consider the following markup:

```
<div class="Dialog">
    <div class="DialogHeader">
        <h2>Popup title</h2>
    </div>
    <div class="DialogBody">
        <p>Popup message</p>
    </div>
    <div class="DialogFooter">
        <p><small>Optional footer content</small></p>
    </div>
</div>
```

This is the markup for the content of a Dialog widget, which pops up to present the user with critical information or to demand mandatory user input. The markup in this example is simplified, but the structure of popup dialogs can grow to become pretty complex. So the HTML author has opted to break up the content of the Dialog widget into three separate widgets, each specialising in presenting the content of a distinct sub-section. Those child widgets are called DialogHeader, DialogBody and DialogFooter. Here are the selectors:

```
.Dialog {}

.DialogHeader {}
.DialogHeader h2 {}

.DialogBody {}
.DialogBody p {}

.DialogFooter {}
.DialogFooter p {}
```

Because the sub-widgets are encapsulated in their own namespaces, each can be developed, tested and maintained separately of the other sub-widgets. And each widget specialises in just one thing. The parent Dialog widget would be responsible only for wrapping everything up in a box and positioning it on the page. Each of the other three widgets need only to focus on presenting the content in one section of a dialog window.

The widgets Dialog, DialogHeader, DialogBody and DialogFooter are clearly related. But there's nothing stopping us from combining entirely unrelated widgets to compose complex UI designs. Here's the markup for a tabbed interface that is made from lots of different things:

```
<div class="Tabs">
    <ul>
        <li><a href="#panel1" class="tabs-selected">Popular</a></li>
        <li><a href="#panel2">Just in</a></li>
        <li><a href="#panel3">Editor's choice</a></li>
    </ul>
</div>
<div class="Panel" id="panel1">
    <h2>Most popular stories</h2>
    <div class="Slats">
        <div class="Slat">
            <h3><a href="#">This is the title</a></h3>
            <p>Lorem ipsum dolor sit amet...</p>
            <p class="meta">10 January, 2015</p>
        </div>
        <div class="Slat">
            <h3><a href="#">This is the title</a></h3>
            <p>Lorem ipsum dolor sit amet...</p>
            <p class="meta">10 January, 2015</p>
        </div>
        <div class="Slat">
            <h3><a href="#">This is the title</a></h3>
            <p>Lorem ipsum dolor sit amet...</p>
            <p class="meta">10 January, 2015</p>
        </div>
    </div>
</div>
<div class="Panel panel-is-closed" id="panel2">
```

```
    ...
</div>
```

In the next example, a grid system (a widget) is used to arrange in rows and columns the content of numerous other nested widgets, all contained within a section of the wider layout called "MAIN".

```
<main class="MAIN">
    <div class="Grid">
        <div class="grid-row">
            <div class="grid-column-one-of-two">
                <div class="Promotion">
                    ...
                </div>
             </div>
            <div class="grid-column-one-of-two">
                <div class="PullQuote">
                    ...
                </div>
             </div>
        </div>
        <div class="grid-row">
            <div class="grid-column-two-of-four">
                <div class="Box box-shade1">
                    ...
                </div>
             </div>
            <div class="grid-column-one-of-four">
                <div class="Box box-shade2">
                    ...
                </div>
             </div>
            <div class="grid-column-one-of-four">
                <div class="Box box-shade3">
                    ...
                </div>
             </div>
        </div>
    </div>
</main>
```

Another methodology for composing complex UIs is to have widgets extend from other widgets. Here's the markup for a widget called Box. It's role is simple: to present a short message in a box. There is an optional button that when clicked will hide the box.

```
<div class="Box">
    <h2>Message</h2>
    <p>Description</p>
    <button>Action</button>
</div>
```

Here are the selectors we'd need to style this widget:

```
.Box {}
.Box h2 {}
.Box p {}
.Box button {}
```

Now imagine that we wanted to modify the default presentation of boxes when the message is particularly important or requires feedback from the user. One option would be to add a modifier class (this technique is covered in the next chapter):

```
<div class="Box box-alert">
    <h2>Message</h2>
    <p>Description</p>
    <button>Action</button>
</div>
```

Another option is to create a new widget that inherits the default presentation of the Box widget. Here's how that can be done:

```
<div class="Box">
    <h2>Message</h2>
    <p>Description</p>
    <button>Action</button>
</div>

<div class="Box BoxAlert">
    <h2>Message</h2>
    <p>Description</p>
    <button>Action</button>
</div>
```

In the markup, notice that the BoxAlert widget (the second one) also uses the encapsulation class for the Box widget. Thus it will inherit the styles for the Box widget and then modify and extend those itself.

Widget inheritance works great when two or more widgets share a similar HTML structure. You might go as far as deliberately making abstract widgets that are not intended to be used on their own, but which provide a basic foundation from which to build other widgets.

Widget inheritance can be convenient, but it can also get messy. Deep levels of inheritance tend to make things harder to maintain, not less so. Try to stick with just a single tier of inheritance. And prefer to err on the side of composition over inheritance. It's generally much easier to compose complex interfaces by combining lots of smaller components with no dependencies of their own.

Modifiers

When something needs to be presented slightly differently than normal, give it a modifier class. Modifiers are the fourth and final component of the Systematic Web Design methodology.

Modifiers are written full lower case with words delimited by hyphens: `"is-required"`, `"sidebar-is-hidden"` and `"contextmenu-selected"`, for example. The first letter must be an alphanumeric character; hyphen prefixes (`"-modifier"`) are illegal, as they are reserved for browser vendors.

Modifier classes should be highly specialised, each setting just a few narrow properties. As a general (but not absolute) rule, it should not be necessary to apply modifiers to achieve a default presentation. Modifiers should be optional. They should only tweak and extend the default presentation of something. A document stripped of all its modifier classes should render perfectly well. There are a few special cases; you might need to apply the clearfix hack, via a modifier called `"clearfix"`, to achieve the desired default presentation, for example.

Try to avoid using modifiers to reset properties inherited from an element's defaults. Ideally, both widgets and modifiers should progressively enhance the default presentation of HTML elements, not regressively degrade them. There is obviously a balance to be found. Set too many properties directly to naked element types, and you might find yourself needing to unset lots of inherited properties when elements are used in special contexts. Set too few default properties, and you will more likely need to maintain a large battery of widgets and modifiers to build out from that minimalist baseline design.

High-fidelity designs – where there is lots of detail and variation in the presentation of things – will inevitably end up with lots of modifiers littered about the HTML. That's okay. The convenience of modifiers in such designs makes up for any difficulties in maintaining neat and tidy markup.

Modifiers can be used in lots of different contexts. A modifier could be:

– Global.
– Specific to one or more element types.

– Specific to a widget.
– Specific to certain elements within a widget.
– Specific to a section of the layout.

And some of those modifiers may be dynamic, swapped in and out of the markup by client-side scripting.

Global modifiers

Global modifiers are specialist classes that are intended to be used almost anywhere. State your intention and include the universal selector (*) when defining global modifiers.

```
*.error { color: red; }
```

Be judicious in your use of global modifiers. They are inherently leaky and may get added to things for which they were not designed, possibly leading to unexpected presentational effects. And rulesets for naked elements and widgets will often have higher specificity, overriding the styles of global modifiers when they are applied to the same elements and widgets. And that in turn may encourage use of the nasty `!important` keyword to give the properties of global modifiers greater precedence. This keyword ought to be used only where you want to ensure that a property cannot be overridden by the client's own style sheets.

```
*.error { color: red !important; }
```

For these reasons, wherever practical prefer to remove modifiers from the global space and instead qualify to what element types, layout sections or widgets they can be applied. This will boost the specificity of the modifiers, eliminating the need for the `!important` keyword and reducing the likelihood of mis-use of the classes.

```
p.error, ul.error,
input.error {
    color: red;
}
```

That's not to say that global modifiers are not useful. They certainly are useful for general utilities like the clearfix hack and for applying niche presentational effects like drop shadows and rounded corners that may recur on different, unrelated components of an interface.

```
<div class="Popup clearfix box-shadow rounded-corners">
```

Element modifiers

HTML's native elements do not always give designers the variety they crave. We often want to introduce variations in the default presentation of individual elements, without consideration to the specific contexts in which those elements are used. Modifier classes that are designed to vary the default presentation of HTML elements should be qualified against the relevant types.

```
p {
    font-size: 1.4rem;
}

p.standout {
    font-size: 1.8rem;
    font-weight: bold;
}

p.smallprint {
    font-size: 1rem;
}
```

Widget modifiers

The default presentation of widgets can be tweaked and extended by application of additional modifier classes. Widget modifiers should be prefixed with the name of the widget that they modify, converted to lower case.

```
.Dialog {
    border: 1px solid hsl(0, 0%, 60%);
    left: 50%;
    margin-left: -400px;
    margin-top: -40px;
    padding: 2rem;
    position: absolute;
    top: 50%;
    width: 800px;
}

.Dialog.dialog-alert { border-color: hsl(15, 100%, 50%); }
```

Usage:

```
<div class="Dialog dialog-alert">
```

Widget modifiers may change the default presentation of the widget's container element, or of any nested elements that form the widget's structure. In the following example the `border-color` property will apply only to the container element for each Dialog instance while the `font-weight` property will be inherited by all of the widget's descendant elements.

```
.Dialog.dialog-alert {
    border-color: hsl(15, 100%, 50%);
    font-weight: bold;
}
```

Modifiers may be needed to target specific instances of an element *within* a widget structure, too.

```
<div class="Dialog dialog-alert">
    <div class="dialog-header>
        <h2>...</h2>
        <p>...</p>
    </div>
    <div class="dialog-body>
        ...
    </div>
    <div class="dialog-footer>
        <button>...</button>
    </div>
</div>
```

Here are the selectors we might use to target each element that composes this widget:

```
.Dialog {}
.Dialog .dialog-header {}
.Dialog .dialog-header h2 {}
.Dialog .dialog-header p {}
.Dialog .dialog-body {}
.Dialog .dialog-footer {}
.Dialog .dialog-footer button {}
```

The prefix convention for widget modifiers is important. The point is to reduce the chances of the same class name being used for two unrelated things, which might cause unexpected property leakages. Consider the following example.

```
<div class="Headline">
    <p class="tagline">Tagline</p>
    <h1>Headline</h1>
    <p class="teaser">Teaser</p>
    <p class="meta">Author, Date</p>
</div>
```

Here are the selectors we'd need to style Headline widget instances:

```
.Headline {}
.Headline h1 {}
.Headline p {}
.Headline p.tagline {}
.Headline p.teaser {}
.Headline p.meta {}
```

There are three modifier classes here: `"tagline"`, `"teaser"` and `"meta"`. The problem with these modifiers is that there is some risk of cross-contamination. It is entirely possible that the following selectors may exist some place else in our style sheets.

```
p.tagline {}
p.teaser {}
p.meta {}
```

Or:

```
*.tagline {}
*.teaser {}
*.meta {}
```

Any properties set by these selectors would be inherited by the Headline widget. The purpose of prefixing widget modifiers is to avoid exactly this scenario.

```
<div class="Headline">
    <p class="headline-tagline">Tagline</p>
    <h1>Headline</h1>
    <p class="headline-teaser">Teaser</p>
    <p class="headline-meta">Author, Date</p>
</div>
```

Here are the final selectors:

```
.Headline {}
.Headline h1 {}
.Headline p {}
.Headline p.headline-tagline {}
.Headline p.headline-teaser {}
.Headline p.headline-meta {}
```

Now there is much less risk that the presentation of the Headline widget will be contaminated by other things. This naming convention has another benefit: it makes it easier to see in the markup which modifiers are global and which belong to particular widgets.

We could now optimise some of our selectors for the Headline widget. After all, the `"headline-"` prefix used for this widget's modifier classes provides adequate encapsulation on its own, does it not?

```
.Headline {}
.Headline h1 {}
.Headline p {}
p.headline-tagline {}
p.headline-teaser {}
p.headline-meta {}
```

Personally, I prefer to always include the parent widget class when selecting child classes.

```
.Headline {}
.Headline h1 {}
.Headline p {}
.Headline p.headline-tagline {}
.Headline p.headline-teaser {}
.Headline p.headline-meta {}
```

This is clearer, consistent and a little more robust. And this is how it would be compiled by a preprocessor if you were using an extended CSS syntax to nest a widget's selectors.

```
.Headline {

    h1 {}

    p {
        &.headline-tagline {}
        &.headline-teaser {}
        &.headline-meta {}
    }

}
```

But try to avoid combinator selectors that are lengthier than this. Selectors should not need more than one descendant level. Prefer to create new classes than to found selectors on the structure of a widget's markup. Things like immediate child selectors and pseudo classes (e.g. `.Headline > h1` and `:first-of-type`) are best avoided, too. They create a tight coupling between HTML and CSS, making it that bit harder to extend and modify widgets in the future.

Layout modifiers

Modifiers may be used also to tweak the default presentation of sections of the layout, too. Layout modifiers should use the same naming convention as for widget modifiers, taking the name of their parent layout section (written lower case) as a prefix.

```
<header class="BANNER banner-homepage">
```

In the CSS, layout modifiers should be further qualified against the class name of the layout section that they modify. This just locks things down that little bit more, and it's not going to be such a big issue if a layout modifier is accidentally used in the wrong place.

```
.BANNER {
    background: hsl(0, 0%, 93%);
    display: table-cell;
    padding: 2rem 0;
    vertical-align: middle;
}

.BANNER.banner-homepage { height: 100vh; }
```

Dynamic modifiers

Widgets may be progressively enhanced with dynamic behaviours. Imagine a widget called Slideshow that displays a series of images. By default, when the HTML is initially rendered, all of the images in a slideshow might be visible, presented one after the other in a vertical layout. The markup might look something like this:

```
<div class="Slideshow">
    <a href="sydney.html">
        <img src="sydney.png" alt="Sydney" />
    </a>
    <a href="melbourne.html">
        <img src="melbourne.png" alt="Melbourne" />
    </a>
    <a href="perth.html">
        <img src="perth.png" alt="Perth" />
    </a>
    <a href="adelaide.html">
        <img src="adelaide.png" alt="Adelaide" />
    </a>
    <a href="darwin.html">
        <img src="darwin.png" alt="Darwin" />
    </a>
</div>
```

Then a JavaScript program called Slideshow.js could come along and dynamically add some classes, which the CSS uses to restyle and animate the slideshow. This is what the markup might look like at one moment in time:

```
<div class="Slideshow slideshow-is-animating">
    <a href="sydney.html" class="slideshow-is-hidden">
        <img src="sydney.png" alt="Sydney" />
    </a>
    <a href="melbourne.html" class="slideshow-is-visible">
        <img src="melbourne.png" alt="Melbourne" />
    </a>
    <a href="perth.html" class="slideshow-is-hidden">
        <img src="perth.png" alt="Perth" />
    </a>
    <a href="adelaide.html" class="slideshow-is-hidden">
        <img src="adelaide.png" alt="Adelaide" />
    </a>
    <a href="darwin.html" class="slideshow-is-hidden">
        <img src="darwin.png" alt="Darwin" />
    </a>
</div>
```

Dynamic modifiers – any class that is dynamically switched in and out of a web page by a client-side JavaScript program – should be prefixed with `"is-"` followed by a verb or noun that describes the component's current state. Temporary classes used to transition something from one state to another are named `"is-"` followed by a verb. Permanent classes that fix the state of something are named `"is-"` followed by an adjective.

```
.is-opening {}
.is-open {}
.is-closing {}
.is-closed {}
```

Dynamic modifiers that belong to a particular widget should be further prefixed with the widget's name (written lower case) and encapsulated in the widget's namespace.

```
.Slideshow.slideshow-is-animating {}
.Slideshow a.slideshow-is-visible {}
.Slideshow a.slideshow-is-hidden {}
```

Classes added to a document for the purpose of feature detection should be prefixed with `"supports-"`.

```
.supports-cssboxsizing {}
.supports-csstransforms3d {}
.supports-webworkers {}
```

The popular Modernizr library can be configured to add a custom prefix to the classes that it injects into documents. Use the "`className prefix`" option when you download the library from here: http://modernizr.com/download/

Classes are often used as targets for DOM queries, too:

```
document.querySelectorAll( '.popup' );
```

Personally, I prefer to use custom `data-*` attributes for this purpose. Using `data-*` attributes – and `<data>` elements, sometimes – as the exclusive interface between HTML and JavaScript leaves class attributes as the primary interface between HTML and CSS. This is a convenient dichotomy, and it is one less concern for classes.

```
document.querySelectorAll( '[data-popup="true"]' );
```

Class names

In the Systematic Web Design methodology, classes are forked into three separate concerns. Each class fulfils one – *and only one* – of the following roles:

– It sets the position of something, helping to create a **layout**.
– It encapsulates the presentation of a discrete UI **widget**.
– It **modifies** the default presentation of something.

Layout classes are applied to sectioning elements: `<main>`, `<header>`, `<footer>`, `<article>`, `<aside>`, `<address>`, `<nav>` and `<section>`, as well as general divider (`<div>`) elements. Layout classes are responsible for setting out the position of content on the page. They do not style content directly. They are concerned exclusively with creating an empty wireframe-like structure in which content – styled separately – can be placed and moved around.

Widgets are custom UI components made from multiple standard HTML elements. Examples include navigation bars, modal popups, accordions, carousels and clusters of social sharing buttons. Optionally, dynamic behaviours may be applied to widgets via JavaScript enhancements. In some cases widgets may be entirely generated by in-browser scripting.

Modifiers are any classes that modify the default presentation of something. A modifier may be global, applicable to just about anything. Or it may be designed to change the default presentation of certain HTML elements, or a certain widget, or even a particular section of the layout. Modifiers may also get added to a document by a client-side JavaScript program to act as targets for CSS animations and transitions and other dynamic presentational effects.

Without exception, every class must either represent the name of a layout section, or the namespace for a widget, or it must exist to modify the default presentation of something. No one class must fuse these separate concerns.

Different naming conventions are used for layout classes, widget classes and modifier classes, to tell them apart. Layout classes are written full upper case, with words delimited by underscores: `"NAV_MAJOR"`, for example. Widget classes use the camel case naming convention:

"NavBar", for example. And modifiers are written full lower case with words delimited by hyphens: "box-shadow", for example.

– Layout: "UPPER_CASE"
– Widgets: "CamelCase"
– Modifiers: "lower-case"

Modifiers that are dynamically added to a document by a client-side script are prefixed "is-" followed by a noun or verb: "is-collapsed", "is-animating", for example. And modifiers that belong to a particular widget or layout section are further prefixed with the name of that widget or layout section, written lower case: "logo-homepage", "contextmenu-is-open", for example.

All three naming conventions can be seen in the following example, which is the markup for a Trello-like project management application.

```
<main class="BOARD">
    <div class="List">
        <h2 class="list-header">
            To do
            <svg><title>Options</title>...</svg>
        </h2>
        <div class="Card card-is-new">
            <a href="card/1234">
                <h3 class="plaintext">Homepage changes</h3>
                <ul class="card-members">
                    <li title="Kieran Potts">KP</li>
                    <li title="Maja Sienkiewicz">MS</li>
                </ul>
            </a>
        </div>
        <div class="Card">
            ...
        </div>
    </div>
    <div class="List">
        ...
    </div>
</main>
```

We can work out from their names and use of capitalisation what each class does. We know that the class "BOARD" positions a section of the main layout, and that "List" and "Card" are the names of widgets – recurring patterns of content – that are reused again and again.

We can also deduce that the modifier class `"list-header"` belongs to the List widget, while `"card-members"` belongs to the Card widget. The `"card-is-new"` modifier also belongs to the Card widget but it got put there by a JavaScript plugin that is responsible for adding dynamic behaviours to any instances of the Card widget that appear in the markup. And we might guess that the `"plaintext"` modifier makes the `<h3>` element look more like a standard paragraph, and that because this class is not prefixed it might work on other element types, too.

What's nice about this naming convention is that the hierarchy of classes is represented by their letter case. Layout classes, written with full capitalisation, scream out from the surrounding content ("HELLO_WORLD"!). The camel case names of widgets, which always will be nested within layout sections, are a little less prominent ("HelloWorld"). And modifier classes, which are arguably the least important classes since they merely modify things and are not critical to achieving a default presentation, are the least conspicuous of all ("hello-world").

Let's see what you make of the classes in the following HTML snippet. Can you guess what each one does?

```
<nav class="NAVIGATION navigation-homepage">
    <div class="BRANDING">
        <div class="Logo">
            <a href="./" class="img"><svg>...</svg></a>
        </div>
    </div>
    <div class="NAV_MAJOR">
        <div class="NavBar">
            <ul>
                <li class="navbar-selected"><a href="learn/">Learn</a></li>
                <li><a href="about.html">About</a></li>
            </ul>
        </div>
    </div>
    <div class="NAV_MINOR">
        <div class="NavBar navbar-minor">
            <ul>
                <li><a href="login.html">Login</a></li>
                <li class="navbar-primary"><a href="join/">Register</a></li>
            </ul>
        </div>
    </div>
    <div class="SEARCH">
        <div class="SearchBox">
            <form action="search.html" method="get">
                <label for="input-search">Search</label>
                <input name="q" type="search" id="input-search" />
                <button type="submit">Search</button>
            </form>
        </div>
        </div>
    </nav>
```

Reading through the markup we know immediately which classes are used for layout, which act as containers for recurring widgets, and which modify the default presentation of an individual element, a whole widget, or even a whole layout zone. These are the layout classes:

- `"NAVIGATION"`
- `"BRANDING"`
- `"NAV_MAJOR"`
- `"NAV_MINOR"`
- `"SEARCH"`

These are the classes that are used to encapsulate distinct globs of content and functionality ("widgets").

- `"Logo"`
- `"NavBar"` (used twice)
- `"SearchBox"`

And these are the modifier classes:

- `"navigation-homepage"`
- `"navbar-selected"`
- `"navbar-minor"`
- `"navbar-primary"`
- `"img"`

It's quite easy to guess what each of those modifiers does. `"navigation-homepage"` obviously changes the default presentation of the `"NAVIGATION"` section of the layout, but only on the website's homepage. `"navbar-selected"` indicates the current selection in a NavBar widget instance. `"navbar-minor"` modifies the default presentation of the second NavBar instance, while `"navbar-primary"` makes the "Register" link more prominent than the "Login" one.

And "img" is a global modifier that is intended to be applied to any anchor element that encapsulates an <img>, presumably to add or subtract certain default properties of anchor elements in that scenario.

Here are the CSS selectors that we could use to style this markup (we might not actually need all of these):

```
.NAVIGATION {}
.NAVIGATION.navigation-homepage {}
.BRANDING {}
.NAV_MAJOR {}
.NAV_MINOR {}
.SEARCH {}

a.img {}

.Logo {}
.Logo a {}
.Logo svg {}

.NavBar {}
.NavBar ul {}
.NavBar li {}
.NavBar li.navbar-selected {}
.NavBar li.navbar-primary {}
.NavBar a {}

.NavBar.navbar-minor {}
.NavBar.navbar-minor ul {}
.NavBar.navbar-minor li {}
.NavBar.navbar-minor li.navbar-selected {}
.NavBar.navbar-minor li.navbar-primary {}
.NavBar.navbar-minor a {}

.SearchBox {}
.SearchBox form {}
.SearchBox label {}
.SearchBox input[type="search"] {}
.SearchBox button {}
```

Class name semantics

Choosing names for things is not easy. In fact it is one of the hardest parts of scalable web design. Names should last the lifetime of the objects to which they are given. So we should choose names that describe things that are unlikely to change. The trick is to pick names that are a little bit abstract, that do not *precisely* describe the content that they encapsulate, or the functionality that they represent, or the presentation that they apply, because these things are inclined to change over time.

Consider the following example. It's the markup for a widget that presents a scrolling marquee of breaking news headlines.

```
<div class="????????">
    <ul>
        <li><a href="story1.html">Headline #1</a></li>
        <li hidden><a href="story2.html">Headline #2</a></li>
        <li hidden><a href="story3.html">Headline #3</a></li>
        <li hidden><a href="story4.html">Headline #4</a></li>
        <li hidden><a href="story5.html">Headline #5</a></li>
    </ul>
    <button type="button">Previous</button>
    <button type="button">Pause</button>
    <button type="button">Next</button>
</div>
```

But what to call this widget? We could describe its presentation, something like `"TopRedBox"`. Classes have no semantic meaning, so visually-descriptive class names are perfectly valid. But are they sensible? What if the colour or position of the box changes?

We could choose a name that describes the content being encapsulated. `"BreakingNews"`, say. But what if the website's news editor decided to use the widget to draw the user's attention to a competition, or a popular blog post? It's conceivable, no? If classes are too literal, their name may become increasingly anachronous to the content they encapsulate as a website evolves over time.

A more generic name like `"TickerTape"` would likely have greater longevity.

```
<div class="TickerTape">
    <ul>
        <li><a href="story1.html">Headline #1</a></li>
        <li hidden><a href="story2.html">Headline #2</a></li>
        <li hidden><a href="story3.html">Headline #3</a></li>
        <li hidden><a href="story4.html">Headline #4</a></li>
        <li hidden><a href="story5.html">Headline #5</a></li>
    </ul>
    <button type="button">Previous</button>
    <button type="button">Pause</button>
    <button type="button">Next</button>
</div>
```

There is a balance to be found. As a general rule, the class names for layout sections and widgets should accurately describe the content that they encapsulate without being too explicit: `"SIDEBAR"` is better than `"RELATED_LINKS"`, for example. A little bit of abstraction, even ambiguity, can be beneficial for these types of classes. But modifier classes, which are usually pretty specialised and apply only a narrow range of properties, tend to benefit from more expressive names: `"rounded-corners"` and `"is-animating"`, for example.

Filesystem

Systematic Web Design is a collection of ideas about how to go about applying CSS to HTML. It is a methodology, nothing more. How you integrate this methodology into your existing workflow is left up to you. But it is conventional in big projects to organise discrete units of development CSS into separate files, and then aggregate and minify everything into a single production-optimised resource. This just makes it easier to navigate the codebase and allows multiple people to work on the CSS in parallel. Task runners and preprocessors automate the assembly process.

This is how I organise my development CSS. I maintain one style sheet for my browser resets and HTML element defaults, and a second style sheet for global and element-specific modifier classes. I have one file for the default layout, and a separate file for each layout variation that extends from the default. And I have one CSS file for each widget. Modifiers that are applicable to specific layout sections or widgets are defined in the most relevant file.

Embedded fonts, keyframe animations, and pseudo elements such as selected text (`::selected`) and scrollbars (`::-webkit-scrollbar`) I tend to keep in the elements style sheet. But in big projects I might move these to separate files for "fonts", "animations" and "chrome". And if I make use of extended CSS syntax I will have a style sheet for "variables" and another for "mixins".

```
@charset "UTF-8";

@import "_fonts";
@import "_animations";
@import "_chrome";

@import "_elements";
@import "_modifiers";

@import "layout/_default";
@import "layout/_HOME";
@import "layout/_SEARCH";
@import "layout/_PRODUCT";
@import "layout/_BASKET";
@import "layout/_CHECKOUT";
```

```
@import "widgets/_Grid";
@import "widgets/_FlyOut";
@import "widgets/_Hello";
@import "widgets/_NavBar";
@import "widgets/_Overlay";
@import "widgets/_Popup";
```

As per convention, imported partials that are not meant to be compiled directly are prefixed with an underscore.

The source order of the compiled CSS *does* matter. Global dependencies ought to come first, with more isolated components such as widgets coming last. This ordering means that the chronology of selectors reflects how things inherit and extend from one another. For example, widget modifiers ought to take precedence over global modifiers, so defining them in this order helps avoid specificity wars.

The ordering of widgets should not matter, but it is convenient that widgets be listed alphabetically and this also ensures that child widgets (e.g. `"DialogAlert"`) always follow their parents (e.g. `"Dialog"`), which they override. For the same reason, page-specific layouts should come after the default layout.

I think most fronties would agree that keeping media queries alongside the base selectors that they extend is the simplest solution to implementing responsive design. Personally, I prefer the mobile-first approach whereby the default presentation of HTML elements, the layout, and individual widgets is optimised for mobile then progressively enhanced for larger viewports and different types of output device.

Mobile-first web design makes extensive use of `min-width` (and sometimes `min-height`) media query parameters. In the following example the padding and margins set via the `"ArticleBox"` class are steadily increased as the viewing window gets bigger. And on screens wider than 1200px (75rem), boxed content is floated to the right edge of its container.

```
.ArticleBox {
    border: 1px solid hsl(0, 0%, 80%);
    margin: 1rem 0;
    padding: 0.5rem 1rem;
}

@media screen and (min-width: 56.25rem) {

    .ArticleBox {
        margin: 2rem 0;
        padding: 1rem 2rem;
    }
}

@media screen and (min-width: 75rem) {

    .ArticleBox {
        border-width: 2px;
        clear: right;
        float: right;
        margin: 0 0 5rem 2rem;
        padding: 2rem 4rem;
    }

}
```

The reverse approach – designing for desktop first then trying to get things to collapse down to mobile – uses `max-width` and `max-height` parameters instead.

```
.ArticleBox {
    border: 2px solid hsl(0, 0%, 80%);
    clear: right;
    float: right;
    margin: 0 0 5rem 2rem;
    padding: 2rem 4rem;
    width: 30%;
}

@media screen and (max-width: 75rem) {

    .ArticleBox {
        border-width: 1px;
        float: none;
        margin: 2rem 0;
        padding: 1rem 2rem;
        width: auto;
    }

}

@media screen and (max-width: 56.25rem) {

    .ArticleBox {
        margin: 1rem 0;
        padding: 0.5rem 1rem;
    }

}
```

Notice how much more code there is. Because desktop designs tend to be more complicated than mobile ones, we have to unset a whole bunch of stuff to get the desktop view to collapse nicely to mobile. The mobile-first approach is much more convenient. We make greater use of the default presentation of HTML elements and we only ever add stuff that is not going to be reset again.

The mobile-first approach works beautifully in all modern browsers. Legacy browsers that do not support media queries (Internet Explorer < 9) will render the default mobile view.

If there is significant variation in presentation between devices, I might break up my master style sheet into several smaller ones, each targetting different clients. This makes things a bit harder to maintain, but the advantage is that no browser is downloading more than it needs to.

```
<link href="mobile.css" media="screen" rel="stylesheet" />
<link href="desktop.css" media="screen and (min-width:800px)" rel="stylesheet" />
<link href="print.css" media="print" rel="stylesheet" />
```

The mobile-first, progressive enhancement approach to web design brings a whole bunch of other benefits. It means that smaller devices – which typically have the slowest processors - have only to render the simplest versions of our web pages. Mobile devices can ignore a whole bunch of stuff in our CSS files. Mobile-first design also forces us to place content in its logical order in the HTML document and not have the source order dictated by the desire to achieve a particular layout on big screens. This improves accessibility. And the additional constraints of mobile devices – screen size, processing power, bandwidth – require that we focus on the stuff that really matters: content, performance, usability and accessibility. Visual pyrotechnics should be at the bottom of our list of priorities, actually.

Thank you

Thank you for reading my book.

If you would like to know when this book is updated, or when I publish new books and related open source projects, please sign up to receive my email newsletter. The registration form is on my website:

http://www.kieranpotts.com

References

AMCSS – Attribute Modules for CSS
http://amcss.github.io
by Glen Maddern and Ben Schwarz

Atomic CSS / Atomizer
http://acss.io
by Yahoo / Thierry Koblentz

Atomic Design
http://bradfrostweb.com/blog/post/atomic-web-design/
by Brad Frost

Beyond Media Queries
http://www.sitepoint.com/beyond-media-queries-time-get-elemental/
by Richa Jain

Block Element Modifier
http://bem.info
by Yandex

BEM 101
https://css-tricks.com/bem-101/
by Robin Rendle, Joe Richardson, *et al*

BEMIT: Taking the BEM Naming Convention a Step Further
http://csswizardry.com/2015/08/bemit-taking-the-bem-naming-convention-a-step-further/
by Harry Roberts

Buffer's CSS
http://blog.brianlovin.com/buffers-css/
by Brian Lovin

Bugsnag's CSS Architecture
https://bugsnag.com/blog/bugsnags-css-architecture
by Max Luster

CodeGuide
http://codeguide.co
by Mark Otto

CodePen's CSS
http://codepen.io/chriscoyier/blog/codepens-css
by Chris Coyier

CSS Guidelines
http://cssguidelin.es
by Harry Roberts

CSS-Tricks Sass Style Guide
https://css-tricks.com/sass-style-guide/
by Chris Coyier

csstyle
http://www.csstyle.io
Sass/PostCSS-based methodology by Dave Geddes

How we do CSS at Ghost
http://dev.ghost.org/css-at-ghost/
by Paul Davis

Github's CSS
http://markdotto.com/2014/07/23/githubs-css/
by Mark Otto

Google HTML/CSS Style Guide
http://google-styleguide.googlecode.com/svn/trunk/htmlcssguide.xml

CSS at Groupon
http://mikeaparicio.com/2014/08/10/css-at-groupon/
by Mike Aparicio

Idiomatic CSS
https://github.com/necolas/idiomatic-css
by Nicolas Gallagher

Interoperable CSS
http://glenmaddern.com/articles/interoperable-css
by Glen Maddern

CSS at Lonely Planet
http://ianfeather.co.uk/css-at-lonely-planet/
by Ian Feather

Multilayer CSS
http://operatino.github.io/MCSS/en/
by Mail.ru

Medium's CSS is Actually Pretty F*ing Good**
https://medium.com/@fat/b8e2a6c78b06

Mobile First
http://abookapart.com/products/mobile-first
by Luke Wroblewski

Modular CSS Naming Conventions
http://thesassway.com/advanced/modular-css-naming-conventions
by John W. Long

Object-Oriented CSS Framework
http://oocss.org
by Nicole Sullivan

OOCSS + Sass
http://ianstormtaylor.com/oocss-plus-sass-is-the-best-way-to-css/
by Ian Storm Taylor

Primer CSS Guidelines
http://primercss.io/guidelines/
by Github

Responsive Web Design
http://abookapart.com/products/responsive-web-design
by Ethan Marcotte

Sass-Based CSS Architecture and Methodology
http://docssa.info
by Matthieu Larcher and Fabien Zibi

Sass Guidelines
http://sass-guidelin.es
by Hugo Giraudel

Scalable and Modular Architecture for CSS
http://smacss.com
by Jonathan Snook

Scaling Down the BEM Methodology for Small Projects
http://www.smashingmagazine.com/2014/07/17/bem-methodology-for-small-projects/
by Maxim Shirshin

SUIT CSS
http://suitcss.github.io
by Nicolas Gallagher

Trello's CSS
http://blog.trello.com/?p=2227
https://gist.github.com/bobbygrace/9e961e8982f42eb91b80
by Bobby Grace