# Scalable and Modular Architecture for CSS

A flexible guide to developing sites small and large.

by Jonathan Snook

SMACSS: Scalable and Modular Architecture for CSS
http://smacss.com

ISBN 978-0-9856321-0-6

Snook.ca Web Development, Inc.
Ottawa, Ontario, Canada
http://snook.ca

Second edition

# Module Rules

As briefly mentioned in the previous section, a Module is a more discrete component of the page. It is your navigation bars and your carousels and your dialogs and your widgets and so on. This is the meat of the page. Modules sit inside Layout components. Modules can sometimes sit within other Modules, too. Each Module should be designed to exist as a standalone component. In doing so, the page will be more flexible. If done right, Modules can easily be moved to different parts of the layout without breaking.

When defining the rule set for a module, avoid using IDs and element selectors, sticking only to class names. A module will likely contain a number of elements and there is likely to be a desire to use descendent or child selectors to target those elements.

Module example

```
.module > h2 {
    padding: 5px;
}

.module span {
    padding: 5px;
}
```

## Avoid element selectors

Use child or descendant selectors with element selectors if the element selectors will and can be predictable. Using `.module span` is great if a span will predictably be used and styled the same way every time while within that module.

Styling with generic element

```
<div class="fld">
    <span>Folder Name</span>
</div>

/* The Folder Module */
.fld > span {
    padding-left: 20px;
    background: url(icon.png);
}
```

The problem is that as a project grows in complexity, the more likely that you will need to expand a component's functionality and the more limited you will be in having used such a generic element within your rule.

Styling with generic element

```
<div class="fld">
    <span>Folder Name</span>
    <span>(32 items)</span>
</div>
```

Now we are in a pickle. We don't want the icon to appear on both elements within our folder module. Which leads me to my next point:

*Only include a selector that includes semantics.* A span or div holds none. A heading has some. A class defined on an element has plenty.

Styling with generic element

```
<div class="fld">
    <span class="fld-name">Folder Name</span>
    <span class="fld-items">(32 items)</span>
</div>
```

By adding the classes to the elements, we have increased the semantics of what those elements mean and removed any ambiguity when it comes to styling them.

If you *do* wish to use an element selector, it should be within one level of a class selector. In other words, you should be in a situation to use child selectors. Alternatively, you should be extremely confident that the element in question will not be confused with another element. The more semantically generic the HTML element (like a span or div), the more likely it will create a conflict down the road. Elements with greater semantics like headings are more likely to appear by themselves within a container and you are more likely able to use an element selector successfully.

## New Contexts

Using the module approach also allows us to better understand where context changes are likely to occur. The need for a new positioning context, for example, is likely to happen at either the layout level or at the root of a module.

## Subclassing Modules

When we have the same module in different sections, the first instinct is to use a parent element to style that module differently.

Subclassing

```
.pod {
    width: 100%;
}
.pod input[type=text] {
    width: 50%;
}
#sidebar .pod input[type=text] {
    width: 100%;
}
```

The problem with this approach is that you can run into specificity issues that require adding even more selectors to battle against it or to quickly fall back to using `!important`.

Expanding on our example pod, we have an input with two different widths. Throughout the site, the input has a label beside it and therefore the field should only be half the width. In the sidebar, however, the field would be too small so we increase it to 100% and have the label on top. All looks well and good. Now, we need to add a new component to our page. It uses most of the same styling as a `.pod` and so we re-use that class. However, this pod is special and has a constrained width no matter where it is on the site. It is a little different, though, and needs a width of 180px.

Battling against specificity

```
.pod {
    width: 100%;
}
.pod input[type=text] {
    width: 50%;
}
#sidebar .pod input[type=text] {
    width: 100%;
}

.pod-callout {
    width: 200px;
}
#sidebar .pod-callout input[type=text],
.pod-callout input[type=text] {
    width: 180px;
}
```

We are doubling up on our selectors to be able to override the specificity of `#sidebar`.

What we should do instead is recognize that the constrained layout in the sidebar is a subclass of the pod and style it accordingly.

Battling against specificity

```
.pod {
    width: 100%;
}
.pod input[type=text] {
    width: 50%;
}
.pod-constrained input[type=text] {
    width: 100%;
}

.pod-callout {
    width: 200px;
}
.pod-callout input[type=text] {
    width: 180px;
}
```

With sub-classing the module, both the base module and the sub-module class names get applied to the HTML element.

Sub-module class name in HTML

```
 <div class="pod pod-constrained">...</div>
<div class="pod pod-callout">...</div>
```

Try to avoid conditional styling based on location. If you are changing the look of a module for usage elsewhere on the page or site, sub-class the module instead.

To help battle against specificity (and if IE6 isn't a concern), then you can double up on your class names like in the next example.

Subclassing

```
.pod.pod-callout { }

<!-- In the HTML -->
<div class="pod pod-callout"> ... </div>
```

You may be concerned about this, depending on the order of loading. For example, on Yahoo! Mail, we have code coming from different places. We had our base button styles and then we had a special set of buttons for the compose screen. However, when you clicked to add a contact to your address book, it loaded a component from a different product: Address Book. (Yes, the address book is a different product within Yahoo!.) The address book loaded its own base button styles, thereby overwriting the sub-classed button styles that we had.

If load order is a factor in your project, watch out for specificity issues.

While more specific layout components assigned with IDs could be used to provide specialized styling for modules, sub-classing the module will allow the module to be moved to other sections of the site more easily and you will avoid increasing the specificity unnecessarily.