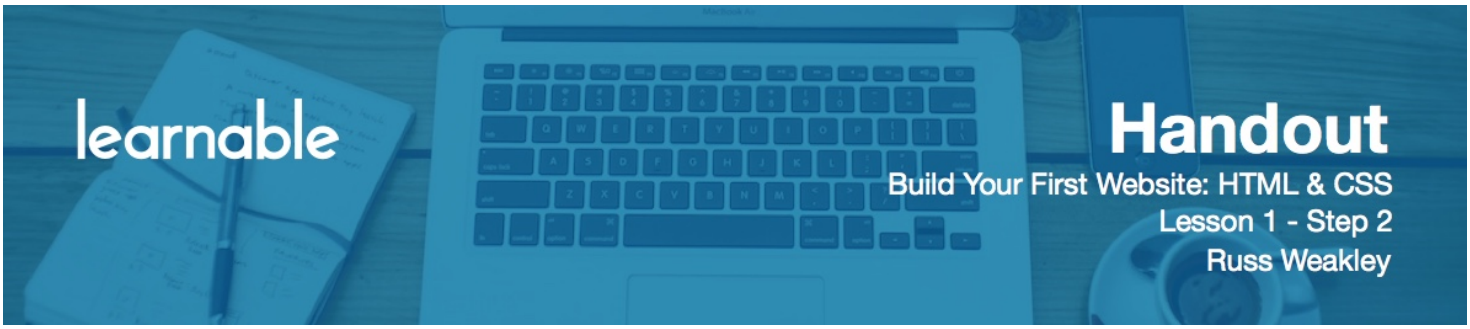

Introduction to moving away full page layouts

In this lesson we are going to explore new methodologies. The first thing we're going to look at is the trend of moving away from pages. A few years ago, many front end developers approached web sites and web applications as a series of pages. These pages were often built as complete entities which meant that the page components were often closely tied to their relevant pages. More recently, the focus has shifted from full page layouts to **reusable modules**. Examples of a reusable module include layout grid structure, a button, an input, or even something like a pull-quote.

HTML/CSS pattern libraries are used to define commonly used interface modules which can easily be re-used as needed. We're going to use this reasonable modules principal to create our own mini-pattern library. The process will involve these steps:

- Looking for modules that may be repeated within the layout.
- Define their characteristics.
- Define class names for them.
- Create the HTML and CSS patterns for those modules.

Introduction to CSS class naming conventions

One of the critical components is the **class name**. It would be best to use a naming convention that will make our class names easier to write during development as well as easy to understand when maintaining the site in the future.

For example, how can we let other team members know that all of the other classes relate to the primary `callout-box` class?

```
<div class="callout-box">
    <h3 class="heading">
    </h3>

    <div class="content">
    </div>
</div>

<div class="callout-box callout-variation">
    <h3 class="heading">
    </h3>

    <div class="content">
    </div>
</div>
```

As you can see we have a class called `callout-box`. We could use a naming convention that was originally defined as part of **BEM (Block Element Module)**. This was then extended by Nicolas Gallagher, and then modified slightly again by Harry Roberts. The core principles state that class names should be

- Consistent
- Relatable
- Recognizable

CSS class patterns - Modules

The primary class for any pattern is called a **module**. When creating CSS class names, I try to avoid `CamelCase`, to make class names as easy to read as possible. Multiple words are joined with a single hyphen. So, for example

```
.calloutBox {}
```

turns to

```
.callout-box {}
```

CSS class patterns - Module Modifiers

If a module needs to be modified or extended, a **module modifier** would be used. Because single hyphens are used for module class names, we're going to differentiate our modifier class names by employing two hyphens. So our module would be

```
.module-name {}
```

and module modifier

```
.module-name--modifier {}
```

You don't have to use this naming convention, but I found it much easier to be able to distinguish different types of classes.

Returning to the initial example:

```
<div class="callout-box">
    <h3 class="heading">
    </h3>

    <div class="content">
    </div>
</div>

<div class="callout-box callout-box--help">
    <h3 class="heading">
    </h3>

    <div class="content">
    </div>
</div>
```

CSS class patterns - Module Descendants

Next let's talk about **module descendants** that can be used when a module has child elements that need to be styled. Use two underscores to make it clear that this is a descendant class:

```
/* Module descendant */
.module-name__descendant-name {}
```

Let's modify the initial examples once again:

```
<div class="callout-box">
    <h3 class="callout-box__heading">
    </h3>

    <div class="callout-box__content">
    </div>
</div>

<div class="callout-box callout-box--help">
    <h3 class="callout-box__heading">
    </h3>

    <div class="callout-box__content">
    </div>
</div>
```

It is easy to see the descendents of the `callout-box`.

But why use class names for descendant elements when we could use descendant selectors instead? We could do something like

```
/* Module descendant selector */
.module-name h3 {}
```

which will style `h3` inside the module with a class `module-name`. However if we use descendant selectors, we're relying on the HTML remaining the same, which may not be the case. Using class names gives us greater stability and flexibility. Jonathan Snook calls this **decoupling** the HTML and CSS.

Extending CSS naming conventions

This basic naming convention can be extended to descendant modifiers as well, though these types of class names would rarely be needed. You could end up having something like

```
/* Module descendant modifier */
.module-name__descendant-name--modifier-name {}
```

however very rarely I need to use these types of class names. You could also create class names that are descendants of descendants, but in my opinion this becomes too complex and too confusing. For really complex modules, it may be much easier to break them down into smaller modules so that naming conventions can remain simple.

Introduction class names used in the course website

Let's have a quick look at the classes that we're actually going to be using in our lessons. First of all, we have a `header` module with some descendants:

```
.header {}
.header__nav {}
.header__nav-about {}
.header__nav-contact {}
```

Then there is `banner-content` module:

```
.banner-content {}
.banner-content__heading {}
```

`features` module:

```
.features {}
.features__row {}
.features__padding {}
.features__heading {}
.features__text {}
.features__bike {}
.features__phone {}
.features__safe {}
```

`testimonials` module:

```
.testimonials {}
.testimonials__heading {}
.testimonials__quote {}
.testimonials__source {}
```

`facts` is a bit more complicated so we've break it into three modules:

```
.facts {}

.facts-intro {}
.facts-intro__heading {}
.facts-intro__text {}

.facts-list {}
.facts-list__chart {}
.facts-list__eye {}
.facts-list__heading {}
.facts-list__icons {}
.facts-list__id {}
.facts-list__text {}
.facts-list__timer {}
```

Next up is the `press`:

```
.press {}
```

```
.press__heading {}
.press__logos {}
.press__logos--bevel {}
.press__logos--nc {}
.press__logos--solvable {}
.press__logos--waratah {}
```

Next is the `footer`

```
.footer {}
.footer__copyright {}
.footer__cta {}
.footer__info {}
.footer__logo {}
```

For `row` we have

```
.row {}

.row--white {}
.row--blue {}
.row--dark-grey {}
.row--grey {}

.row--padding-medium {}
.row--padding-wide {}

.row-banner {}
```

There are also a couple of places where there is no primary modules, just a series of options instead. A classic example of this is `container`:

```
.container-narrow {}
.container-medium {}
.container-wide {}
```

There is no default container to modify, so we're just going for `container-narrow`, `container-medium` and `container-wide`. Similarly with the columns:

```
.col-narrow {}
.col-medium {}
.col-wide {}

.col-narrow--right {}
.col-wide--right {}
```

There are also a few small areas where I have broken my own rule and used some descendant selectors:

```
.header__nav {}

.header__nav ul {}
.header__nav li {}
.header__nav a {}
```

A separate class could be assigned for these `ul`, `li` and `a`, but because they're pretty unique, I didn't think that it is really necessary.

Similarly, the `footer`:

```
.footer__info {}

.footer__info ul {}
.footer__info a {}
```

And finally `footer__copyright`:

```
.footer__copyright {}

.footer__copyright li {}
```

Despite these last few examples, almost all of our CSS contains class name selectors without descendant selectors. This makes it more efficient for browsers, and easier for maintenance.