STANDARDIZING YOUR CODE

Working on projects with more than one person – or even projects that you hope others might review, means agreeing on (and sticking to!) a pattern for naming your styling elements, logic variables, and file names.

When you start work at a new company, one of the first things you should learn is what naming conventions they follow.

BEM

What is it?

BEM stands for **Block-Element-Modifier** and refers to a naming convention for your CSS classes. It was developed at a company called Yanex starting in 2005. (https://en.bem.info/method/history/)

It focuses on identifying and labelling the major components of your web page (Blocks) and their corresponding sub-components (Elements & Modifiers). It is similar in nature to the CSS parent/child relationship, or the html <div> and <section> elements.

A **Block** is a, "…logically and functionally independent page component…" (https://en.bem.info/method/key-concepts/), or, a distinct section of your web page. The header would be a Block, the footer would be a Block, the sidebar would be a Block, the social media area would be a Block, and the navigation bar in the header would be a Block within a Block. Essentially, anything that might use a <div> or <section> would be considered a Block. As a guideline, if it could be moved to a different spot of the page without breaking the page, or could be duplicated on other pages, it is probably a Block.

An **Element** is one piece of a block that belongs to that block and would not appear outside of that block. An **`<input/>`** element is always part of a form (or, an input area) and would not be used outside of that area.

A **Modifier** is a version of an Element that is different in appearance. Going back to our form example, a required **`<input/>`** field might be styled differently from one that was not required; or we might want to make our default **`<option>`**

look differently from the other **<option>** Elements. These would be variations, or modified versions of the input Elements.

How do I use it?

BEM is implemented simply by identifying and labelling your HTML elements as major Blocks; sub Elements dependent on the parent Block; or Modifiers of Elements. Let's get to an example:

Naming the Block. If you use camelcase (camelCase) to name your class, BEM asks you to use a dash delimiter between words. So, **`userForm`** becomes **`user-form`**. *HOWEVER* - JavaScript doesn't like dashes in variable names (it thinks it is a minus operator!), so if you are going to use the id/class in JavaScript you should *avoid* using the dash and stick to camel case.

Naming the Element: To name an Element, use the parent Block's name, followed by *two underscores*, then, the Element name: **`user-form__email`**.

Naming Modifiers: Add a Modifier with a *single underscore* after the Block/Element before the Modifier name: **`user-form__email_invalid`**.

It's that simple. Look at the following examples, and you can see how following BEM leads to easy-to-read class names that also convey a hierarchical or semantic meaning.

`cstmrInfoIdHidden`	VS	**`cstmr-info__id_hidden`**
`signupFormBtnSubmit`	VS	**`signup-form__btn_submit`**
`signupFormBtnClear`	VS	**`signup-form__btn_clear`**

Why use it?

- Easier for others to read your code – it creates an intuitive pattern.
- Demonstrates production experience and understanding.
- By demonstrating your ability to follow a styling standard it increases the perception of you as a professional.

For more information:

https://en.bem.info/method/key-concepts/
https://css-tricks.com/bem-101/
http://getbem.com/introduction/

http://csswizardry.com/2013/01/mindbemding-getting-your-head-round-bem-syntax/
http://getbem.com/naming/
https://8thlight.com/blog/nelsol-batalla/2014/08/01/bem-basics.html
https://medium.com/fed-or-dead/battling-bem-5-common-problems-and-how-to-avoid-them-5bbd23dee319#.h5378dyxm

"use strict";

What is it?

"use strict"; is a directive that you add to your JavaScript code indicating that the browser should execute your code following a subset of JavaScript introduced in ECMAScript 5.

What does it do?

It asks the browser to enforce a stricter set of rules on your JavaScript, throwing syntax errors that would not be thrown in regular JavaScript. This might not sound desirable, but the type of errors it catches can be considered sloppy coding that might lead to unpredictable behaviour in your code.

Why use it?

It can prevent global variable errors that you would not catch otherwise, and it shows an intermediate/advanced level of understanding to those viewing your code.

How do I use it?

There are two ways to use it: on the global level, and within an individual function.

If you wish to declare it on a global level, simply make **`"use strict";`** the first line of your JavaScript file. Declaring at a global level will affect your entire file. If you wish to only implement it on a per function basis, then make the declaration the first line of your function:

```
function doStuff(){
    "use strict";
    var stuffValue = true;
    return stuffValue;
}
```

If I use it, what do I have to do differently?

So, what are these "stricter rules"? For the most part, it means that **you must declare all variables with a *var* statement** before you use them. If you don't, it will throw an error. The second major restriction is that you cannot write to a read-only or get-only property. Additionally, there are some new reserved words and a few methods that are considered dodgy that you are restricted from using: **`with`**, **`eval`** and **`delete`** are banished.

If you are considering implementing it, start by coding normally, then add the use strict directive once you've finished and see if that throws any errors. Note that if you declare strict on a global level in one script, then concatenate another script onto it, the strict rules will be applied to it as well.

For more information:

W3 Schools has a brief overview: http://www.w3schools.com/js/js_strict.asp, and a great article with more detail about why you should use it can be found at: https://www.nczonline.net/blog/2012/03/13/its-time-to-start-using-javascript-strict-mode/ .

JSLint

What is it?

JSLint is a JavaScript program that analyzes and reports on your JavaScript code in much the same way that the W3C Validator evaluates your HTML and CSS. It is important to know that, like "use strict", it uses a subset of JavaScript as well as standardized conventions for the styling of your JavaScript code. For example:

- The keyword *function* should always be followed by one space.
- Any operators should have one space around them: **`2+3`** should be **`2 + 3`**.
- The first line inside of a code block should be indented by four spaces (not one tab!)

JSLint is also particular about certain coding practices, **`+= 1`** is required where **`++`** would be used.

Rather than list all of the requirements, the best way to learn is to scan through the help page at http://www.jslint.com/help.html , then load up one of your labs from last semester and read the error messages, correcting them one at a time.

Why use it?

Like "use strict" the intention of this subset is to prevent logic errors that would not cause a runtime error. Additionally, following this standard shows an intermediate/advanced level of programming – or least an awareness of (and adherence to) a logical and stylistic standard i.e. a professional, and not an amateur level of coding.

How do I use it?

The easiest way is to go to jslint.com and paste in your code.

Some IDE's and script runners have JSLint installed to check your code as you work. To install the JSLint package (which runs every time you save your code) in Sublime:
(Windows) https://packagecontrol.io/installation#st2
(Mac) https://opensourcehacker.com/2012/04/12/jslint-integration-for-sublime-text-2/

For more information:

jslint.com
http://www.jslint.com/help.html