FRONT END WEB DEVELOPMENT:

LESSON 19: PREPROCESSORS

JAMES WILLOCK, SOFTWARE ENGINEER
GENERAL ASSEMBLY

Objectives

- Preprocessors
  - HAML
  - SCSS
  - CoffeeScript

PREPROCESSORS

Preprocessors

- In computer science, a preprocessor is a program that processes its input data to produce output that is used as input to another program
- Preprocessed languages are ones which "compile down" to another language
  - HAML – or HTML Abstraction Markup Language – compiles to HTML
  - SCSS compiles to CSS
  - CoffeScript compiles to JavaScript
- Front-end preprocessed languages are not understood by web browsers, and must be processed first!

Uses

- Preprocessed languages typically add more "syntactic sugar" to make a language more pleasant to write
- Often, a preprocessed language will add constructs outside of the definition of the language it compiles to
  - SCSS adds variables, mixins, nesting and user-defined functions to CSS
  - CoffeeScript allows you to write JavaScript in a classically orientated way – JavaScript is a prototypical language

Popularity

- Preprocessors are becoming popular as many web frameworks are coming bundled with support by default
  - Ruby on Rails supports SCSS and CoffeeScript by default
- Many front-end frameworks are written using preprocessed languages and then distributed in their processed formats
  - Bootstrap is written using LESS and distributed as CSS

HAML

HTML Abstraction Markup Language compiles to HTML and adds makes writing HTML much leaner with emphasis on whitespace

SCSS

Sass is a CSS preprocessor with two syntaxes – SASS and SCSS. SCSS is the more popular option, but both add variables, mixins etc.

COFFEESCRIPT

CoffeeScript compiles to JavaScript and adds the concept of "classes" to writing JavaScript, a prototypical language

Using preprocessors

- The language preprocessing would generally happen on the back-end of an application
- Often, particular during prototyping, front-end developers will use a GUI application to "watch" a directory for changes in preprocessed files and process them automatically
- We'll use prepros, an OS X and Windows application for these exercises

HAML

- The HAML maintainers discuss the following principles for the project:
  - Markup should be beautiful
  - Markup should be DRY
  - Markup should be well-indented
  - Structure should be clear

Indentation

- The most visually obvious difference between writing HAML and HTML is the usage of whitespace to control hierarchy
  - There are no closing tags in HAML; indentation is used to control the hierarchy of tags

HAML

```
!!! 5
%html
  %head
    %title My page
  %body
    %h1 My title
    %p Some content
```

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Page</title>
  </head>
  <body>
    <h1>My title</h1>
    <p>Some content</p>
  </body>
```

Tags

- Tags are written using the percentage symbol and the tag name: `%p`, `%a`, `%head`, etc
- Attributes are passed to tags using a Ruby hash – something like a JavaScript object

```
%a{href: 'http://google.com'} Google
```

- Passing multiple attributes is as simple as comma-delimiting them:

```
%p{id: 'cool', class: 'awesome'}
```

Filters

- HAML also supports filters – a way of breaking out into another language or syntax
- One popular way of writing content is with Markdown, which HAML can yield to

```
%div
  :markdown
    # Headline
    - List item
    - List item
```

CODE ALONG:
WRITING HTML WITH HAML

SCSS

- Sass – Syntactically Awesome StyleSheets – is a CSS preprocessor with two syntaxes, SCSS and SASS
  - SCSS is the most popular format, and is merely a subset of CSS, meaning regular CSS inside SCSS is perfectly valid
  - SASS is a “leaner” syntax, using whitespace for hierarchy in a similar way to HAML

Variables

- Variables will eventually make their way into the standard CSS language specification, but until then we can use them in SCSS
- Give us an easy way to make one definition and copy its result in other places in our stylesheet

CSS

```
body {
  background: #7FDBFF;
  font: 100%/1.4 Georgia, serif;
}

footer {
  border: 1px solid #7FDBFF;
}

a {
  color: #7FDBFF;
}
```

SCSS

```
$light-blue: #7FDBFF;
$serif-stack: Georgia, serif;

body {
  background: $light-blue;
  font: 100%/1.4 $serif-stack;
}

footer {
  border: 1px solid $light-blue;
}

a {
  color: $light-blue;
}
```

Nesting

- Nesting is a natural way to denote the hierarchy of styles without repeating yourself
- It can get a bit messy, so don't over-use it!

CSS

```css
nav {
  background: red;
}

nav ul {
  list-style: none;
}

nav ul li {
  float: left;
}

nav ul li a {
  background: blue;
}
```

SCSS

```scss
nav {
  background: red;

  ul {
    list-style: none;

    li {
      float: left;

      a {
        background: blue
      }
    }
  }
}
```

Mixins

- Mixins allow us to group a set of styles together and "mix them in" to other selectors to afford DRYer code
- A mixin call also take arguments (these are called *parametric mixins*) to behave somewhat like a function

Mixins

- Mixins allow us to group a set of styles together and "mix them in" to other selectors to afford DRYer code
- A mixin call also take arguments (these are called *parametric mixins*) to behave somewhat like a function

```scss
@mixin button($color) {
  border-radius: 4px;
  padding: 10px 15px;
  background: $color;
  color: white;
}

.primary-cta {
  @include button(#000);
}

.secondary-cta {
  @include button(#333);
}

.twitter-button {
  @include button(#0074D9);
  float: left;
  width: 200px;
}
```

CODE ALONG:
WRITING CSS WITH SCSS

COFFEESCRIPT

Principles

- CoffeeScript attempts to “expose the good parts of Javascript”
- Like HAML, uses whitespace for program flow
- Adds some niceties to the language, like string interpolation, classes and natural English expressions
- The var keyword is gone and CoffeScript automatically scopes variables for you, as well as wrapping everything in an anonymous function to prevent polluting the global scope

Functions + interpolation

JavaScript

```
var sayName = function(name) {
  return "Your name is " + name + "!";
}

console.log(sayName("James"));
# Your name is James!
```

CoffeeScript

```
sayName (name) ->
  "Your name is #{name}"

console.log sayName("James")
# Your name is James!
```

Functions + interpolation

```
class ScreamingButton
  constructor: (@button) ->

    @button.on 'click', =>
      this.scream()


  scream: ->
    @button.append(" Aaaaah!")

$ ->
  primary = $('.primary-cta')
  new ScreamingButton(primary)
```

Expressions

JavaScript

```
if (passwordTrue && age >= 18) {
  console.log("Come in!");
} else {
  console.log("Go away!");
}
```

CoffeeScript

```
if passwordTrue and age >= 18
  console.log "Come in!"
else
  console.log "Go away!"
```

CODE ALONG:

WRITING JAVASCRIPT WITH COFFEESCRIPT

THANK YOU

- james.willock@generalassemb.ly
- @niceguyjames