**SASS (Syntactically Awesome Style Sheets)**

**Advantages of SASS:**

* CSS COMPATIBLE
* FEATURE RICH: Sass boasts more features and abilities than any other CSS extension language out there.
* LARGE COMMUNITY
* FRAMEWORKS: There are an endless number of frameworks built with Sass. Compass, Bourbon, and Susy just to name a few.

**PREPROCESSING:**

CSS on its own can be fun, but stylesheets are getting larger, more complex, and harder to maintain. This is where a preprocessor can help. Sass lets you use features that don't exist in CSS yet like variables, nesting, mixins, inheritance and other nifty goodies that make writing CSS fun again. Once you start tinkering with Sass, it will take your preprocessed Sass file and save it as a normal CSS file that you can use in your website.

**Variables:**

Think of variables as a way to store information that you want to reuse throughout your stylesheet. You can store things like colors, font stacks, or any CSS value you think you'll want to reuse. Sass uses the $ symbol to make something a variable.

$font-stack: Helvetica**,** sans-serif;

$primary-color: #333;

body {

**font**: 100% $font-stack;

**color**: $primary-color;

}

**NESTING:**

When writing HTML you've probably noticed that it has a clear nested and visual hierarchy. CSS, on the other hand, doesn't. Sass will let you nest your CSS selectors in a way that follows the same visual hierarchy of your HTML.

Ex.

nav {

ul {

**margin**: 0;

**padding**: 0;

**list-style**: none;

}

li { **display**: inline-block; }

a {

**display**: block;

**padding**: 6px 12px;

**text-decoration**: none;

}

}

After converting into css

nav ul {

**margin**: 0;

**padding**: 0;

**list-style**: none;

}

nav li {

**display**: inline-block;

}

nav a {

**display**: block;

**padding**: 6px 12px;

**text-decoration**: none;

}

**PARTIALS:**

You can create partial Sass files that contain little snippets of CSS that you can include in other Sass files. This is a great way to modularize your CSS and help keep things easier to maintain. A partial is simply a Sass file named with a leading underscore. You might name it something like “\_partial.scss”. The underscore lets Sass know that the file is only a partial file and that it should not be generated into a CSS file. Sass partials are used with the @import directive.

**IMPORT:**

CSS has an import option that lets you split your CSS into smaller, more maintainable portions. The only drawback is that each time you use @import in CSS it creates another HTTP request. Sass builds on top of the current CSS @import but instead of requiring an HTTP request, Sass will take the file that you want to import and combine it with the file you're importing into so you can serve a single CSS file to the web browser.

**MIXINS**

Some things in CSS are a bit tedious to write, especially with CSS3 and the many vendor prefixes that exist. A mixin lets you make groups of CSS declarations that you want to reuse throughout your site. You can even pass in values to make your mixin more flexible. A good use of a mixin is for vendor prefixes. Here's an example for transform.

**@mixin** **transform**($property) {

-webkit-transform: $property;

-ms-transform: $property;

**transform**: $property;

}

**.box** { **@include** **transform**(**rotate**(30deg)); }

**EXTEND/INHERITANCE:**

This is one of the most useful features of Sass. Using @extend lets you share a set of CSS properties from one selector to another. It helps keep your Sass very DRY. In our example we're going to create a simple series of messaging for errors, warnings and successes using another feature which goes hand in hand with extend, placeholder classes. A placeholder class is a special type of class that only prints when it is extended, and can help keep your compiled CSS neat and clean.

*// This CSS will print because %message-shared is extended.*

%message-shared {

**border**: 1px solid #ccc;

**padding**: 10px;

**color**: #333;

}

**.message** {

**@extend** %message-shared;

}

**.success** {

**@extend** %message-shared;

**border-color**: green;

}

.error {

@extend %message-shared;

border-color: red;

}

.warning {

@extend %message-shared;

border-color: yellow;

}

When you generate your CSS it will look like this. Note that the CSS in %equal-heightsdoesn't print because it is never used.

**.message,** **.success,** **.error,** **.warning** {

**border**: 1px solid #cccccc;

**padding**: 10px;

**color**: #333;

}

**.success** {

**border-color**: green;

}

**.error** {

**border-color**: red;

}

**.warning** {

**border-color**: yellow;

}

**OPERATORS:**

Doing math in your CSS is very helpful. Sass has a handful of standard math operators like +, -, \*, /, and %. In our example we're going to do some simple math to calculate widths for an aside & article.

**.container** { **width**: 100%; }

article**[**role**=**"main"**]** {

**float**: left;

**width**: 600px **/** 960px **\*** 100%;

}

aside**[**role**=**"complementary"**]** {

**float**: right;

**width**: 300px **/** 960px **\*** 100%;

}

We've created a very simple fluid grid, based on 960px. Operations in Sass let us do something like take pixel values and convert them to percentages without much hassle. The generated CSS will look like:

**.container** {

**width**: 100%;

}

article**[**role**=**"main"**]** {

**float**: left;

**width**: 62.5%;

}

aside**[**role**=**"complementary"**]** {

**float**: right;

**width**: 31.25%;

}

**The Role Attribute defined in this specification allows the author to annotate markup languages with machine-extractable semantic information about the purpose of an element**

**JavaScript Questions**

**Array:**

An array is a special variable, which can hold more than one value at a time.

**Array sort:**

Array sorting for a string works fine but for numbers it doesn’t work fine. However, if numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1". Because of this, the sort() method will produce incorrect result when sorting numbers.

**Date:**

var d = new Date("2015");

When one string argument is given then it will take it as a year but when one number argument is given then it will take it as a milliseconds.

**Short dates are written with an "MM/DD/YYYY" syntax like this:**

var d = new Date("03/25/2015");

**Date.parse() :** If you have a valid date string, you can use the Date.parse() method to convert it to milliseconds**.** Date.parse() returns the number of milliseconds between the date and January 1, 1970:

**When comparing a string with a number, JavaScript will convert the string to a number when doing the comparison. An empty string converts to 0. A non-numeric string converts to NaN which is always false.**

For break and continue: <https://www.w3schools.com/jS/js_break.asp>

A code block is a block of code between { and }.

**A Proper Random Function:**

This JavaScript function always returns a random number between min (included) and max (excluded):

function getRndInteger(min, max) {  
    return Math.floor(Math.random() \* (max - min) ) + min;  
}

This JavaScript function always returns a random number between min and max (both included):

function getRndInteger(min, max) {  
    return Math.floor(Math.random() \* (max - min + 1) ) + min;  
}

**TYPE CONVERSION:**

* **typeof null         // Returns "object"**
* **typeof myCar // Returns "undefined" (The data type of an undefined variable is undefined )**
* **typeof [1,2,3,4] // Returns "object"**
* **typeof new Date() // Returns "object"**
* **typeof function () {} // Returns "function"**
* **typeof NaN // Returns "number"**
* The typeof operator is not a variable. It is an operator. Operators ( + - \* / ) do not have any data type. But, the typeof operator always returns a string (containing the type of the operand).
* The constructor property returns the constructor function for all JavaScript variables.

Ex. [1,2,3,4].constructor

// Returns function Array()   {[native code]}

{name:'John',age:34}.constructor

// Returns function Object()  {[native code]}

new Date().constructor

// Returns function Date()    {[native code]}

* **To Check if a variable is Array or not:**

function isArray(myArray) {  
    return myArray.constructor.toString().indexOf("Array") > -1;  
}

OR

function isArray(myArray) {  
    return myArray.constructor === Array;  
}

* **To Check if a variable is Date or not, we will similar code mentioned above for arrays.**
* JavaScript variables can be converted to a new variable and another data type:
  + By the use of a JavaScript function
  + Automatically by JavaScript itself
* **toFixed()** returns a string, with the number written with a specified number of decimals

Ex. var x = 9.656;  
 x.toFixed(0);           // returns 10  
 x.toFixed(2);           // returns 9.66

* **toPrecision()** returns a string, with a number written with a specified length

var x = 9.656;  
x.toPrecision();        // returns 9.656  
x.toPrecision(2);       // returns 9.7

* Number(" ")       // returns 0   
  Number("")        // returns 0  
  Number("99 88")   // returns NaN

Number(null) // return 0

* **Automatic Type Conversion**

5 + null    // returns 5         because null is converted to 0  
"5" + null  // returns "5null"   because null is converted to "null"  
"5" + 2     // returns "52"      because 2 is converted to "2"  
"5" - 2     // returns 3         because "5" is converted to 5  
"5" \* "2"   // returns 10        because "5" and "2" are converted to 5 and 2

**Automatic String Conversion:** JavaScript automatically calls the variable's toString() function when you try to "output" an object or a variable

**REGEX:** [**https:/www.w3schools.com/js/js\_regexp.asp**](https://www.w3schools.com/js/js_regexp.asp)

* Regular expressions can be used to perform all types of text search and text replace operations.

**JavaScript Errors:**

* The try statement lets you test a block of code for errors.
* The catch statement lets you handle the error.
* The throw statement lets you create custom errors.
* The finally statement lets you execute code, after try and catch, regardless of the result.

The **try** statement allows you to define a block of code to be tested for errors while it is being executed.

The **catch** statement allows you to define a block of code to be executed, if an error occurs in the try block.

try {

Block of code to try

}

catch(err) {

Block of code to handle errors

}

finally {

Block of code to be executed regardless of the try / catch result

}

The **throw** statement allows you to create a custom error. Technically you can throw an exception (throw an error).

The **finally** statement lets you execute code, after try and catch, regardless of the result.

Ex. function myFunction() {  
    var message, x;  
    message = document.getElementById("p01");  
    message.innerHTML = "";  
    x = document.getElementById("demo").value;  
    try {   
        if(x == "") throw "is empty";  
        if(isNaN(x)) throw "is not a number";  
        x = Number(x);  
        if(x > 10) throw "is too high";  
        if(x < 5) throw "is too low";  
    }  
    catch(err) {  
        message.innerHTML = "Error: " + err + ".";  
    }  
    finally {  
        document.getElementById("demo").value = "";  
    }  
}

**THE ERROR OBJECT:**

JavaScript has a built in error object that provides error information when an error occurs.

The error object provides two useful properties: name and message.

Reference error:

var x;  
try {  
    x = y + 1;   // y cannot be referenced (used)  
}  
catch(err) {  
    document.getElementById("demo").innerHTML = err.name;  
}

Result: ReferenceError

**The debugger keyword acts as a breakpoint in the browser debugger.**

**JavaScript Hoisting:**

Hoisting is JavaScript's default behavior of moving declarations to the top i.e a variable can be used before it has been declared. JavaScript Initializations are Not Hoisted. Because of hoisting to avoid bugs, always declare all variables at the beginning of every scope.

x = 5; // Assign 5 to x  
  
elem = document.getElementById("demo"); // Find an element   
elem.innerHTML = x;                     // Display x in the element  
  
var x; // Declare x

**STRICT MODE:**

"use strict"; Defines that JavaScript code should be executed in "strict mode". It helps you to write cleaner code, like preventing you from using undeclared variables. Strict mode makes javascript code more secure.

Declared inside a function, it has local scope (only the code inside the function is in strict mode)

function myFunction() {  
   "use strict";  
    y = 3.14;   // This will cause an error  
}

Deleting a variable (or object) or function is not allowed.

"use strict";  
var x = 3.14;  
delete x;

The string "eval" cannot be used as a variable.

The string "arguments" cannot be used as a variable

For security reasons, eval() is not allowed to create variables in the scope from which it was called.

The "use strict" directive is only recognized at the beginning of a script or a function.

**THIS KEYWORD:**

When used alone or inside function, this refers to the Global object. But inside object or object function it refers to the object.

In a function definition, this refers to the "owner" of the function.

In strict mode, this will be undefined, because strict mode does not allow default binding.

**JAVASCRIPT MISTAKES:**

* JavaScript programs may generate unexpected results if a programmer accidentally uses an assignment operator (=), instead of a comparison operator (==) in an if statement.

This if statement returns true (maybe not as expected), because 10 is true:

var x = 0;  
if (x = 10) // return true;

This if statement returns false (maybe not as expected), because 0 is false:

var x = 0;  
if (x = 0) // return false;

* Confusing Addition & Concatenation:

Addition is about adding numbers. Concatenation is about adding strings. In JavaScript both operations use the same + operator.

var x = 10 + 5;          // the result in x is 15  
var x = 10 + "5";        // the result in x is "105"

* Breaking a Return Statement: It is a default JavaScript behavior to close a statement automatically at the end of a line. Because of this, these two examples will return the same result

function myFunction(a) {  
    var power = 10    
    return a \* power  
}

function myFunction(a) {  
    var power = 10;  
    return a \* power;  
}

* If you use a named index, when accessing an array, JavaScript will redefine the array to a standard object.

var person = [];  
person["firstName"] = "John";  
person["lastName"] = "Doe";  
person["age"] = 46;  
var x = person.length;         // person.length will return 0  
var y = person[0];

Result: person = [firstName: "John", lastName: "Doe", age: 46]

**NEW FEATURES INTRODUCED IN ES6**

* JavaScript let
* JavaScript const
* JavaScript default parameter values
* Array.find()
* Arrat findIndex()

**From JS NUMBERS to JS MATH go through w3schools**