JavaScript Style Guide

For your class, we expect that all students adhere to a few standards relating to syntax, formatting, naming, and terminology. This is a guide for how JavaScript is expected to be written out as well as rules that we will be following in class. Before we dive into the content, it is important to understand that writing code is similar to painting a picture. There many different styles and opinions on best practices; code can be written in many different ways. Since there are so many acceptable applications of JavaScript, we have established this style guide to keep projects at Grand Circus consistent, to reduce the stress of learning all possible JavaScript solutions, and to set expectations for code reviews and general feedback.

What Is Syntax?

Syntax is simply a set of rules that your program needs to follow in order to be constructed properly. JavaScript, and all coding languages, require us to use syntax properly.

Case Sensitivity

JavaScript is case sensitive. You cannot write a program that changes case sensitivity of variables, functions, and keywords whenever you want. This is important to remember. Variables, functions, and other keywords should be lowercase

```
var school = “Grand Circus”;
function sayHello() {
  alert(“Hello”);
}
```

Built-in objects and constructors are capitalized. In the following example, the built-in object known as Number is going to be capitalized.

```
var num = Number(prompt(“Type a number 1 - 10”));
```

Methods are generally lowercase. Most all of the methods that are provided by JavaScript are lowercase. In the following example, we will take a look at the method known as toLowerCase().

```
var name = “Adam”;
console.log(name.toLowerCase());
```

We use a convention known as camel-case to write variable and function names that require more than one word. The idea behind camel-case is it makes variable and function names much easier to read. Take a look at the two functions below and think about which is easier to read.

```
function unittypeconversion() {
  // awesome code goes here
}

function unitTypeConversion() {
```

```
  // more awesome code
}
```

This is going to be very common across all languages, so might as well get used to seeing it now!

```
var userInput = prompt("Enter your name.");
function getUserInfo() {
  // some cool code
}
```

Semi-colons

We expect to use semicolons in all appropriate places. The general rule is you will use a semicolon after a statement. Think of semicolons as periods in the English language. We wouldn't want to start a new line without ending the previous line, right?

Any time we declare a variable, we need to end the statement with a semicolon.

```
var name = "Adam";
var total = 10;
```

Whenever we assign values to a variable, we want to be ending the statement with a semicolon.

```
name = "Grant Chirpus";
total++;
```

Inside for loops, we need to initialize, set a condition, and create final expression. These will be separated by semicolons.

```
for(var i = 0; i <= 10; i++) {
  console.log(i);
}
```

We typically will never want semicolons after curly braces. The only time a semicolon should appear after a closing curly brace is after an object literal is declared and initialized.

```
var userInfo = {
  name: "Grand Circus",
  address: "1570 Woodward Avenue",
  city: "Detroit"
};
```

Spacing

Any spaces or tabs in your code is referred to as white space. White space is completely ignored by the JavaScript compiler. What does that mean? It means Example 1 and Example 2 do the same thing.
Example 1:

```
if (3 + 3 === 6) { console.log(“This is true”); }
```

Example 2:

```
if (3 + 3 === 6) {
  console.log(“This is true”);
}
```

In our class, we will go with Example 2. To start off, it is easier to read, especially when your code base gets much bigger. Generally, all of our slides and demos will be containing a two-space indentation(or a tab set to be two-spaces) following a new line.

If you look at Example 2, you will see that there is a space after the `if` keyword prior to the opening parenthesis of the condition and a single space after the closing parenthesis of the condition prior to the opening curly brace. The same will be true of function and variable declarations. Just

```
var language = “JavaScript”;

function showContent() {
  // do something
}
```

Quotation Marks

JavaScript allows us to use either double or single quotation marks. You can use either set, but you must close a quotation with the same type. For example, if we have a string that is started with a single quotation mark, we must close it with a single quotation mark. If we start a string with a double quotation mark...we must close it with a double quotation mark.

Our slides and demos will be aligned to use double quotation marks.

Where Do I Include My Script Tag?

We expect that all script tags get linked above the closing </body> tag in your HTML document. You may see that some tutorials and documentation will have scripts linked in the <head> of the HTML document. For consistency, at Grand Circus, we link scripts above the close </body> tag.

Examples

Here are some basic examples that follow the naming, spacing, and semicolon rules from above in case you a quick reference.

Variables

```
var name = “Grand Circus”;
var userName = “GrantChirpus1”;
var userList = [“Adam”, “David”, “Yasmine”];
var myInfo = {
  name: “Grant Chirpus”,
  work: “Grand Circus”,
  title: “Mascot”
};
```

Functions

```
function addNumbers(x, y) {
  return x + y;
}
```

For Loop

```
for (var i = 0; i <= 10; i++) {
  console.log(i);
}
```

Do..while Loop

```
var i = 0;

do {
  console.log(i);
  i++;
} while(i <= 10);
```

While Loop

```
var i = 0;
while (i <= 10) {
  console.log(i);
  i++;
}
```

Conditions

```
if (true) {
  // do this
} else if (true) {
  // do this
} else {
  // do this
}
```

jQuery

jQuery provides us with a robust list of features we can use in our projects. Below we will outline how these methods should be constructed and how we expect everyone to write their code. jQuery is just JavaScript, so there aren't a ton of new things here.

Document Ready

jQuery introduces a ready method, which we will use extensively with jQuery. There are two ways to use the ready method but our class will be using just one. The following code is what we expect in every script file that contains jQuery, so get comfortable with it!

```
$(document).ready(function() {

  // all of our code goes in this

});
```

.on() Method

When using jQuery methods, we expect everyone to follow a very simple standard: use the .on() method. Your code should be displayed just like our example

```
$("button").on("click", function() {

  // do something when a button element is clicked

});
```

Notice how we start with our jQuery selector, which is immediately followed by the .on() method. Once again, we are going to be utilizing our normal spacing, normal indentation, and normal semicolon rules.

jQuery events will be done using the parameter "event". Please see the example below

```
$("button").on("click", function(event) {

  // do something when a button element is clicked

  // since we passed in a parameter to the anonymous function, we can now utilize information about the event(time it triggered, what element it was on, sibling or parent elements to the event...

});
```

Angular

Angular is a massive framework. One cohesive guide on just Angular alone could be upwards to 50 pages. We will document important conventions that we expect everyone to adhere to during the bootcamp. Once again, this is just JavaScript; Angular just looks a little different.

The first thing we will discuss is going to be relating to organization of code. Angular is going to require multiple script files, anywhere from 2-10(or more!). It is very important to keep your project organized and modular. To do this appropriately, we are going to require a new script file for **each** component. Yes, that means if your project requires one module, two controllers, and a service...you will have each of those components in their own file.

The second most important topic you should be aware of is naming conventions. Your files should be named after the component. If you have a controller called “homeController”, your file should be “homeController.js”.

Following those two rules will set you up for success. Your code will be easier to maintain, read, and debug.

Before we show demos of each of these components, there is just one more rule we want you to follow. All of your components should be wrapped in IIFE’s(immediately invoked function expression). This will protect your code from any issues with naming collisions or scope when it comes time to minify your code.

Modules

The ng-module directive will be included in the <html> tag.

For creating a module(setter syntax), we will use the following syntax

```
// module.js
(function() {
  angular
    .module(“app”, []);
})();
```

For retrieving a module(getter syntax), we will use the following syntax

```
// in any JavaScript file that isn’t module.js
(function() {
  angular
    .module(“app”);
})();
```

Controllers

The ng-controller directive will be included on your <body> tag, unless you are using routing. If routing is being used, you should include ng-controller on elements that require that controller's logic.

```
// controller file
(function() {
function TestCtrl() {
  var vm = this;
  vm.message = “Hello”;
}

angular
  .module(“app”)
  .controller(“TestCtrl”, TestCtrl);
})();
```

Notice the controller component has two parameters. It has a name for the controller and a constructor function. Both need to be uppercase.

Services

Services are components that do not need to be referenced anywhere other than in a controller. The syntax is very close to all other Angular components, however, different service components are different. Both the name and constructor function need to be uppercase.

Here is an example of a factory, which is one of the service components.

```
// factory file
(function() {
function TestService() {
  return {
    name: name
  };
}
function name() {
  return “My name is Adam”;
}

angular
    .module(“app”)
    .factory(“TestService”, TestService);
})();
```

Here is an example of a service, which is another service component.

```
// factory file
(function() {
function TestService() {
  this.name = function() {
    return “My name is Adam”;
  }
}

angular
  .module(“app”)
  .service(“TestService”, TestService);
})();
```

Components

Components allow us to include markup and a controller within a single object. We can reference this component anywhere within the application. Please note, the “template” property of this object is not in single quotes. It is an ES6 template literal.

```
// component file
(function() {
var OurDirective = {
  bindings: {},
  template: `<somehtml></somehtml>`,
  controller: function() {
    // functionality
  }
}

angular
  .module(“app”)
  .component(“ourComponent”, ourComponent);
})();
```

Once we create the directive, we will include it in our HTML as follows:

<our-component></our-component>

Directives

Directives are going to be the last component we will discuss. Directives are very important to Angular; we need to understand how to create these appropriately.

```
// directive file
(function() {
function OurDirective() {
  return {
    restrict: “AE”,
    replace: false,
    templateUrl: “partials/our-directive.html”,
    link: function($scope, $element, $attrs) {
```

```
      // do something with this directive
    }
  }
}

angular
  .module("app")
  .directive("ourDirective", OurDirective);
})();
```

Once we create the directive, we will include it in our HTML as follows:

If the directive is being used as an element:

<our-directive></our-directive>

If the directive is being used as an attribute:

<div our-directive>