**Exercise to go with Assignment 1**

In this exercise we will play with our own custom modules.

***The first part of this exercise supports the Syed book 'best practices' on pulling together multiple modules (here english.js and spanish.js) in a larger module (index.js) for easier management.***

1. In a convenient place which you can navigate to from your command window make a folder named **project1**.
2. Inside that folder make a folder named **greet**.
3. Using notepad++ write the following 3 files and save them in the **greet** folder.

NOTE: You should not imitate the coloring – I have written variables in red and property values in blue to make it easier for you to understand what is what.

**english.js**

   var greet = function() {

 console.log('Hello');

 }
module.exports = greet;

**spanish.js**

    var greet = function() {

 console.log('Hola');

 }

module.exports = greet;

**index.js**

 var english = require('./english');

var spanish = require('./spanish');

module.exports = {

 english: english,
 spanish: spanish

};

In this code there are 3 JavaScript files in the same **greet** folder. Because they are in the same folder, in index.js we may require them with **require(‘./fileName’);
Please note that (i) we use quote marks around the file name; (ii) we use ./ to indicate we are in the same folder – actually the . says this folder; (iii) each js file uses module.exports to export either a function (in emglish.js and spanish.js) or an object whose properties are functions; (iv) there is an annoying custom of using the same identifier multiple times.**

1. Now write the following code and save it in the **project1** folder as **app.js** var greet =  require('./greet');  //looks for the index.js file in the greet folder

                                           //gets what module.exports says in index.js

 greet.english();    //runs the function associated with the english property of greet

 greet.spanish():   //runs the function associated with the spanish property of greet

 **NOTE:** In this code, there is no greet.js module, so node will look for the **greet** folder and in it will look for **index**.**js**This is actually not a wonderful way to do things - but I wanted you to see the default at work.

1. Next in your command window, navigate over to the **project1** folder and type

 **npm init**Answer the questions, accepting defaults where they are offered.

Go back to your **project1** folder (using Windows Explorer) and you will see that the npm has created a **package.json** file.

Examine the package.json file using VS Code.
2. Next, as a comand type (in the Terminal)

 **node app**
3. **As time allows** go to the Modules page in our course and play with some of the alternate ways to play with these ideas.
4. When we start to require and install (locally) core modules you will see that init also sets up folder for those modules. The node\_modules folder will hold the modules which are dependencies for our app.js

***The second part of this assignment reinforces the material in the Syed book about shared content vs. the factory pattern. Recall that when you require a module M a copy of that module M is cached (& shared across the files in your application.) So that values which M exported are shared. But if the module M you required has exported a function, then each time that function is called you have a new run of it.***

***This is an important and subtle distinction and is reinforced in this code based on the Syed examples.*** *You should make the appropriate folders and files and run the code. In your group you should be sure that you can explain what is happening.*

**Syed examples**

**Syed\_shared folder**

foo.js
module.exports = {

    something: 123

};

app.js

var foo = require('./foo');

console.log('initial something:', foo.something); // 123

// Now modify something:

foo.something = 456;

// Now load bar:

var bas = require('./bar');

bar.js
var foo = require('./foo');

console.log('in another module:', foo.something); // 456

**Run app**

PS C:\myNode\Syed\_shared> node app

initial something: 123

in another module: 456

**Notice that within this folder, foo is shared and so when we chan**g**e the value of something, that change persists, even in other modules (here in bar.js)**

**BUT – if we have foo export a function, then each time the function is called we will instantiate a new object!**

**Syed\_factory folder**

**foo.js - notice that foo now exports a function (not an object)**module.exports = function () {

    return {

        something: 123

    };

};

**app.js which now uses the function that foo exported to create an object**

var foo = require('./foo');

//Note that foo is a function, not an object

//create a new object

var obj = foo();

//use it

console.log(obj.something);

// Now modify something:

obj.something = 456;

// Now load bar:

var bas = require('./bar');

**bar.js also uses the function which foo exported**

var foo = require('./foo');

var newObj = foo();

console.log(newObj.something);

**When we run app - the change in the value of obj.something does not affect the value of what we created in bar.js ---- that is the constructor function is shared, but each time we run it we get another obect!**

PS C:\myNode\Syed\_factory> node app

123

123

**MORAL: When you export an object that object is shared. If one of the properties in the object is a VALUE, then the VALUE is shared across files in that project/folder. When the property is a FUNCTION, then each time you run the FUNCTION you instantiate a new object.**

**Exercise:** Write a new version of foo.js that exports

function() {return
 { aValue:123,
 aFunction: function() {
 console.log(this.aValue;
 return this.aValue} //end of aFunction
 } //end of the object which this function returns
} //end of the anonymous function.

Write new versions of app.js and bar.js which do the following:
 app.js should require foo, and use it to create an object appObj.
 (That is appObj = foo(); )
 It should log the value returned by appObj.aFunction and then change appObj.aValue to 456.
 Finally it should require bar.js (which will run bar.js)
Meanwhile, bar.js should also use the function returned by foo to create barObj = foo().
What will happen when app.js calls bar.js?

**Turning to events:**

1. How do you get ahold of the module which includes events handling?
2. What does that module export?
3. How do you instantiate a new event?
4. How do you assign an event handler to that event?
5. How do you make that event fire and how do you pass extra information to the event handler when it fires?
6. What are two pages which describe the other functions you can perform on events (e. g. remove an event listener or find all the listeners for a particular event?
7. There were several examples of extending events.
You may need to read code which extends events, but you won't be writing such code.
Let's do the one from
<https://www.hacksparrow.com/nodejs/eventemitter-tutorial-examples.html>
	1. In your folder for Node examples create a new folder. I named mine spackRadio.
	2. Copy the code from the radio.js and example.js on that page and save them with those names in your folder.
	3. In VS Code open the terminal (unless it's already open) and navigate to that folder.
	For example, I have navigated to C:\myNode\spackRadio
	4. In the Terminal type
	 npm init
	and answer the questions --- I referenced the web site as the source and hacksparrow as the author.
	Be sure that example.js is the starting point.
	5. In the Terminal type: node example

Sit back and watch.

p.s. if you have trouble reading the code in hacksparrow here it is:

**radio.js**var util = require('util');

var EventEmitter = require('events').EventEmitter;

// @station - an object with `freq` and `name` properties
var Radio = function(station) {

 // we need to store the reference of `this` to `self`, so that we can use the current context
 //in the setTimeout (or any callback) functions

 // using `this` in the setTimeout functions will refer to those funtions, not the Radio class.
 //Instead of declaring self we could use call(this) as in the code after this example, but
 //using self avoids all those scoping issues.

 var self = this;

 // emit 'open' event instantly
 setTimeout(function() {
 self.emit('open', station);

 }, 0);

 // emit 'close' event after 5 secs
 setTimeout(function() {

 self.emit('close', station);

 }, 5000);

 // EventEmitters inherit a single event listener, see it in action
 this.on('newListener', function(listener) {
 console.log('Event Listener: ' + listener);
 });
};

// extend the EventEmitter class using our Radio class|
util.inherits(Radio, EventEmitter);

// we specify that this module is a refrence to the Radio class
 module.exports = Radio;

**example.js**var Radio = require('./radio.js');

// a station object
var station = {
 freq: '80.16',
 name: 'Rock N Roll Radio',
};

// create an instance of the Radio class
var radio = new Radio(station);

// add an 'open' event listener
 radio.on('open', function(station) {
 console.log('"%s" FM %s OPENED', station.name, station.freq);
 console.log('♬ ♫♬');
});

// add a 'close' event listener
radio.on('close', function(station) {
 console.log('"%s" FM %s CLOSED', station.name, station.freq);});

**Extending a module**

There are several steps for this:
I will use the example of a *Cat* extending an *Animal* – because it's easy to keep straight which way the extension goes.

* **Require the util module**const util = require('util');
The utility module has an inherits method which we will use as
 util.inherits(cat, animal);

I think of this from as "inherits from" --- i.e. cat inherits from animal.
That way I can remember that the first parameter is the new object I am creating, and the second parameter is what we are building on.
* **Require the object you wish to extend**
const Animal = require('./Animal'); // more commonly it may be a core module.
* **Define a function for your new object, and inside it call the constructor method of the original** **object --- so that anything defined directly on the old object will also be created on the new object.**
**You may also add other properties:**function Cat(name) {
 Animal.call(this);
 this.name = name;
 this.lives = 9;
* **Use the inherits method to attach the prototype of the ori**g**inal object to the new object:** util.inherits(Cat, Animal);
* **After you have done that you may choose to add more methods to the prototype of the new object. *It must be done afterwards, or the inherits() will kill it.***Cat.prototype.climb() = function() {
 //do stuff
 }
* **Instantiate some examples**myCat = new Cat('Tabby');
Tabby will have all the methods and properties of Animal and of Cat.

If you are feeling ambitious: the code looks slightly diferent is you had defined Animal as a *class*.

A class has a function named *constructor* (you can guess what that does) and if we have required Animal we could code:
 class Cat extends Animal {
 constructor(name ) {
 super(); //this takes care of all the inheritance, including calling Animal's
 //constructor on our new object
 this.name = name;
 this.lives = 9:

 climb(){
 //do stuff
 }
 }

Here is another good example of extending the event emitter functionality: (It may be easier to read if you look at this is landscape orientation):

Clear example of creating your own module which extends eventEmitter

From <https://teamtreehouse.com/community/eventemitter-using-this-in-node-utilinherits-new-profile-many-explanations-missing>

I was having similar concerns when I originally did that course. However, I ended up researching the EventEmitter class of the node core modules and found some cool stuff on it. In this video particularly we are creating our own class 'Profile' which inherits all methods and properties from the node module EventEmitter. We can use event emitter to emit our own custom events in our application. We can then listen to these events and handle them however we want to. I included a basic example (without any error handling stuff) below on how we can create our own EventEmitter and use it in other files.

Imagine this is in file: MyOwnEmitter.js

*//Here at the top of the file you are requiring some core modules from node.js*

*//what it does: it 'imports' some core functionalities from node.js*

*//EventEmitter: The base 'class' for your 'custom' event emitter*

var EventEmitter = require("events").EventEmitter;

*//https: Allows you to make https calls to other resources (here our own made up api)*

var https = require("https");

*//util: is used so you can extend EventEmitter and create your own 'child' class of it.*

var util = require("util");

*//Now create our own 'class constructor'. We do this so later in our application we can instantiate our eventemitter.*

function MyOwnEmitter() {

 *//Call the Node.js EventEmitter constructor (you need to do this so you can inherit from it later)*

 EventEmitter.call(this);

 *//We create a local variable 'self' which just refers to itself so we do not create any scoping issues when doing our api call*

 var self = this;

 *//let's do our api call to some made up api that returns weather data, by using the https module from above.*

 *//first parameter: the url we are requesting. second parameter: a callback function to run after the request is made*

 var request = https.get('https://someMadeUpWeatherApi/', function(response) {

 *//set the body just to an initial string initially*

 var body = "";

 *//Here we listen for the 'data' event and add to the body as the data is coming to our server*

 response.on('data',function(chunk){

 body += chunk;

 *//HERE IS KEY: WE ARE ACTUALLY EMITTING OUR OWN EVENT NOW*

 self.emit('dataFromMyApiIsHere', chunck);

 });

 *//Here we listen for the 'end' event which is fired once our response is done. Which means we can parse the data*

 response.on('end',function() {

 *//parse the body into json so we can easily read it wherever we want to later*

 var json = JSON.parse(body);

 *//HERE IS KEY: WE ARE ACTUALLY EMITTING OUR OWN EVENT NOW AGAIN*

 self.emit('endOfOurApiData', json);

 });

 }

}

*//Here we inherit from EventEmitter so our 'class' MyOwnEmitter can use all of the same methods as EventEmitter. like //'emit' and 'on' as we have already seen above in self.emit etc.*

util.inhertis(MyOwnEmitter, EventEmitter);

*//export our class/module so we can use it in other files*

module.exports = MyOwnEmitter;

Imagine now this is in the file app.js

var myOwnEmitter = require('./MyOwnEmitter.js');

*//now we instantiate our own emitter;*

*//What happens at this moment -- it runs through everything we put in the function MyOwnEmitter above.*

*//Including the api call to our made up weather api*

var myOwnEmitterInstance = new myOwnEmitter();

*//Now we can listen for the events 'dataFromApiIsHere' and 'endOfOurApiData' we emitted above*

myOwnEmitterInstance.on('dataFromApiIsHere',function(chunk){

 console.log(chunck);

});

myOwnEmitterInstance.on('endOfOurApiData',function(json){

 *//this will actually be all of the data our weather api returned in json format!!!*

 console.log(json);

});

**More things to be aware of:**

**Reminder about modules and npm init**

The package.json is created by npm init andit uses anything you have installed so far --- see the dependencies property in package.json

If you add more modules (that you later decide you need) use

***npm install <module name> --save***

Then npm will update the dependencies.

You will also, coutesy of npm init,   have a node\_modules folder created with the relevant modules/packages.

**Deprecated modules and the terminal vs VS Code**

VS Code is very diligent about not running code which uses deprecated modules or functions. The terminal may be more or less helpful. For example, consider this code, which comes from the Syed book (listing 3-31)

//Listing 3-31 from Syed 'Beginning Node.js

var str = "Hello Buffer World";

var buffer = new Buffer(str, 'utf-8');

var roundTrip = buffer.toString('utf-8');

console.log(roundTrip);

In the terminal:



In VS Code:


(Altho' VS Code sometimes shows the same error as the terminal).

The issue is that Node will deprecate modules and /or functions and you may be looking at older code. If you are going to do a lot of work in Node you want to know when there are changes in the core modules --- they are usually for good reasons. *But ALSO, you may want to have your project stick to a specific version of a module, so that updates etc. don't break your code.* Please note that while JavaScript is very, very good about maintaining backwards compatibility, Node.js is much less so.

Take the code in this small app and run it both in your terminal and in VS Code. What do you see?

**More on types of modules**

**Node has 2 types of modules, both fully supported: CommonJS modules and ECMAScript modules.**

**CommonJS modules**<https://nodejs.org/api/modules.html> are the type for all core modules and are the default:

We can **require** them and **export** selected methods and properties.

**ECMAScript modules**<https://nodejs.org/api/esm.html>

are also supported in Node, and are ECMA's official version of modules for packaging and reuse.

We can **import** them (or parts of them) but need the .js extension.  We can  **export**

*Authors can tell Node.js to use the ECMAScript modules loader via the .mjs file extension, the package.json*[*"type"*](https://nodejs.org/api/packages.html#type)*field, or the*[*--input-type*](https://nodejs.org/api/cli.html#--input-typetype)*flag. Outside of those cases, Node.js will use the CommonJS module loader.*

When I turn to the most recent Node documentation, for some reason, it has switched over to the ECMAScript syntax by default:
<https://nodejs.org/api/buffer.html#static-method-bufferfrombuffer>  starts with

  import { Buffer } from 'buffer';

NOTE: Since buffer is a core module the import...'buffer'  replaces the previous require('buffer')

Also, Buffer is now an object named Buffer in this file.

ALSO NOTE: That the button on the right allows you to switch between CommonJS and ECMScript syntax.

If you want to learn more about ECMAScript modules and import, turn to <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import>

The default encoding in buffers is utf-8.

**Automated restartin of servers --- esp. while in development mode.**

There is a package **nodemon** which will automatically restart your server (when we write more server code) when even you make a change to your project. This is a time saver and quite useful. You install it as usual (perhaps globally) and can even restrict it to when you are in developer mode- see <https://stackabuse.com/automatically-restart-node-apps-on-file-change-with-nodemon/> or <https://awik.io/automatically-reload-node-js-app-saving/> Once you install it, instead of typing
**node app** you type ***nodemon app***

**In case you encounter a class**A class is usually considered to be just syntactic sugar for folks who come from the world of Java. It is a way to make a constructor function for some objects (with properties and methods) look more like Java. See <https://javascript.info/class> As you will quivkly figure out, there is no new intellectual content here.

As time allows, (in the console --- no node here) write a class for point, with a method showCoords, instantiate a couple of points, and show their coordinates.