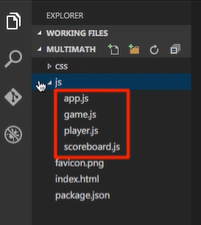
**Into & goals of modularity**

* Create high-level abstractions of functionality and split them out. Modules let us think of chunks of functionality instead of individual functions. You don’t want to worry about brick by brick, but how the building fits together.
* **Encapsulation**: clear API for interaction, but black-box re: implementation. Lets the module owner make huge changes internally without affecting the consumers, because the API is standard.
* **Reusability**: realistic reusability might be modules that can be re-used within a single app, or across a department.
* **Simplify dependency management**: ideally every module explicitly declares which modules it requires. There would be tooling support to pull all of those together when needed. This is how webpack works!

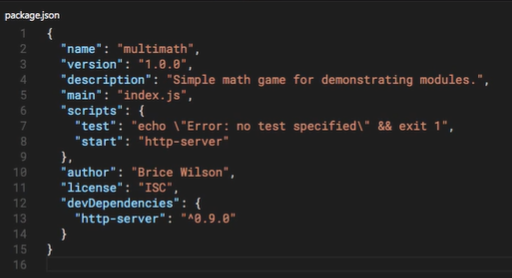
Note: NPM is the package manager included with Node.js. That’s the difference between those two things – Node isn’t a package manager, it just includes one!

**First application – very little complexity**

Four modules.

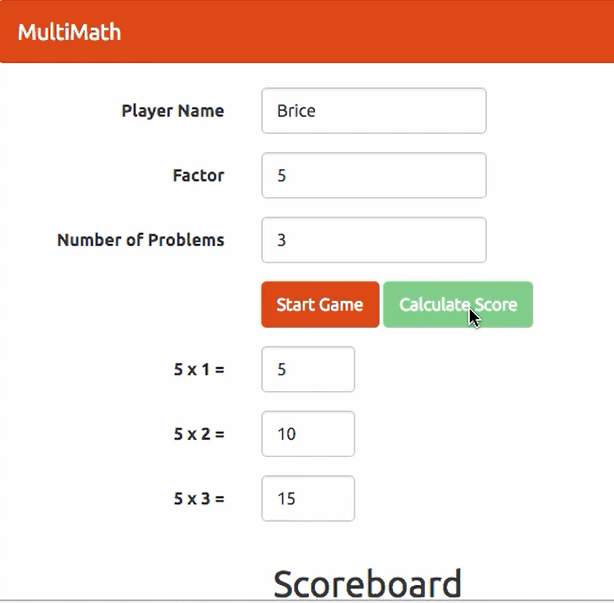
**App** is the main module, it initialises the game.

The game, player, and scoreboard are collections of functionality that combine to make the app work.

package.json has a few simple things in it:

http-server asa dev dependency. He’ll use it to test on during development. There’s a reference to the script he’ll use to run http-server in the ‘scripts’ reference.

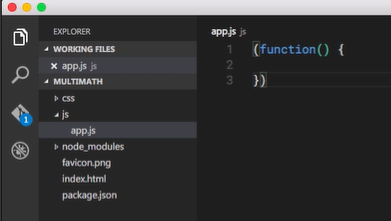
Npm install obviously installs all packages in the package.json file to the node\_modules folder in your root.



**Basic modules: IIFE (Immediately invoked function expressions)**

* An anonymous function invoked when it’s declared. Variables are scoped to functions, and an IIFE is just a function, so you can encapsulate logic within that function knowing it won’t be callable from anywhere else, and because it’s all initialised at runtime, you don’t need to worry about naming clashes (global scope pollution).
* Provide some encapsulation.
* Reduce global scope pollution.
* Note – you’d only really use these if you desperately needed to – they aren’t great!

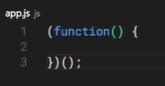
*App.js*



This is a function expression. We’ve got a function with no name (anonymous) and we’ve put brackets around it to tell the parser that this isn’t a new function declaration, but a function expression that should be evaluated. That’s the F and the E parts sorted!

The final part of the IIFE is ‘immediate execution’. So far we’ve got a function, we’ve put it in brackets to say ‘evaluate, this isn’t a declaration’, and now we add two brackets to run it – immediately on parsing! IIFE.

*App.js*

This is the basic structure of an IIFE. As soon as it’s read by the interpreter it’ll be executed immediately.

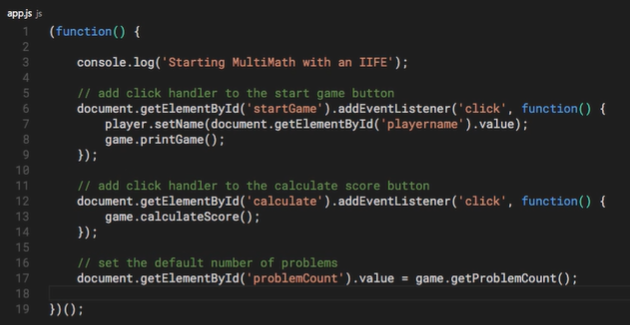
Finally, we import our new IIFE by adding it with a script tag. At the end of our index.html.

*Index.html*



Once we’ve fleshed this out a bit the IIFE module lets us initialise the whole game based on the existing UI elements.

*App.js*



Note that we haven’t implemented ‘player’ or ‘game’ yet. How do we create these and call them from this code? Let’s find out.

Note also that we aren’t returning anything – this function is just anonymously pulled into index.html and executed. In doing so it changes the page, but it’s a very basic, limited form of module.

**Slightly more complex (but still pure JavaScript): The Revealing Module Pattern:**

This takes advantage of vars being scoped to functions in JS. Unlike an IIFE, we need to be able to refer to modules in the code, so they need a name. This one’s clear, simple, separates the module’s private implementation details and public API.

It doesn’t provide any form of dependency management, however – you need to manually determine what’s required for this module and include it.

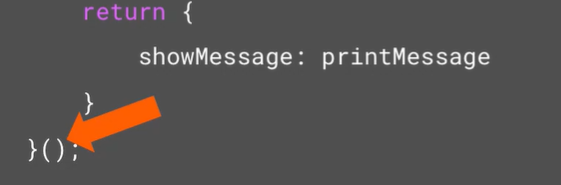
***Note that implementing a module as a singleton means we can only have one instance of that object***. It gets evaluated and run when the parser reaches the JS import statement ‘<script src=…’ in the HTML, and provides access to the singleton object.

**Singleton**:

* One way of implementing an RMP. We define a function and save it to a variable. This returns an object literal.
* The function should return an object that represents your new module. In most cases it exposes functions and properties within the private segment of the module.
* Below is an example – the showMessage property of our returned object actually calls the private ‘printMessage’ function.



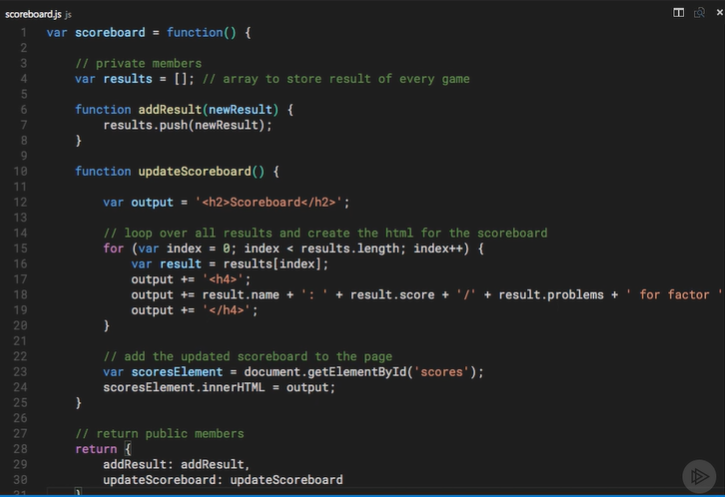
* Note that we’re not just defining a new function, we’re also *immediately calling it with a pair of brackets at the end.* So we aren’t assigning a new function to the variable ‘scoreboard’, we’re assigning the results of that function, an object!



Let’s try a few more singletons.

Here’s our first module:

* Anonymous function that is immediately called with ().
* Has a private var and three functions (get & set), all returned as an object.



And above is our next one: it gives a bunch of functionality around storing and updating the values in the UI. Note how gross this is! This is how I used to write JS, and why react etc. are so important.

Finally, we’ll implement a game module, which will follow the same pattern (anonymous function that is immediately executed and returns an object).

**Revealing Module Pattern: Constructor functions instead of Singletons**

***Note that implementing a module as a singleton means we can only have one instance of that object***. It gets evaluated and run when the parser reaches the JS import statement ‘<script src=…’ in the HTML, and provides access to the singleton object.

We’ll refactor our code to allow it to perform as a constructor.



* We now no longer immediately execute the function – we pass the definition of the function, rather than the result of the code, so instead of an object you get a function.
* To call it we’d do this:



**Section conclusion:**

* Traditionally JS sucks at enabling modules.
* We talked about IIFEs, and how they can be defined.
* We talked about Revealing Module Pattern:
  + Singlestones
  + Constructors
* These give us a nice clean syntax for displaying a public syntax for consuming the module that doesn’t expose its internal implementation details.