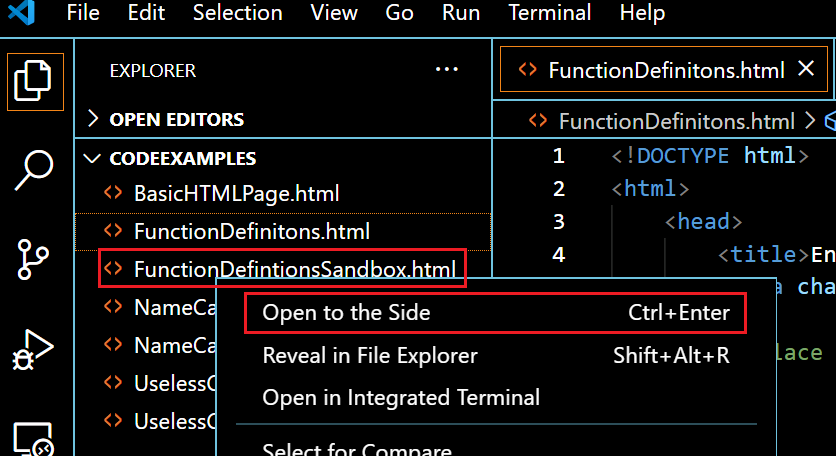
Before finally moving to actual “client / server,” i.e. html request / response, programming I want to spend time on JavaScript functions:

* Callbacks
* Closures
* ES6 methods for functions
* Other interesting things that can be done with them

Functions can be declared in several ways. Note – some of this will be simplistic but we will build on it so bear with me.

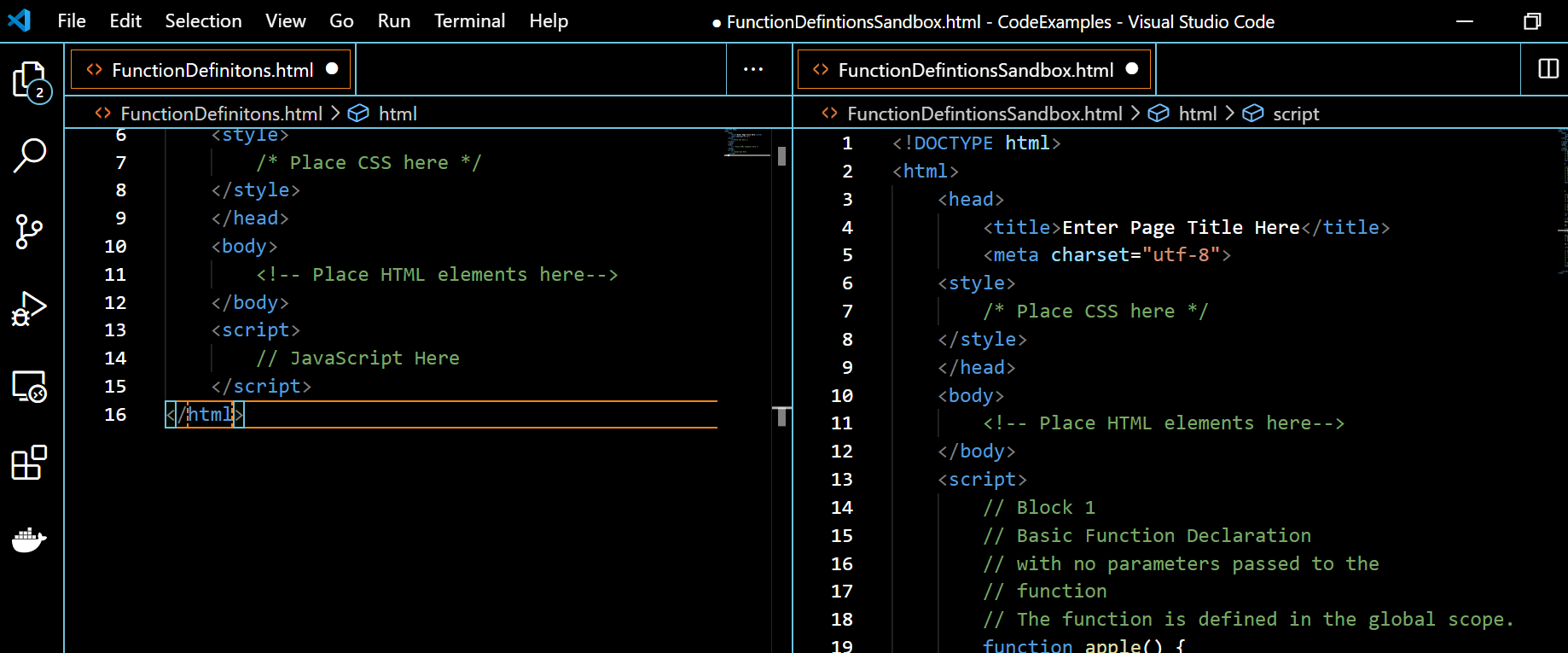
Open the file “FunctionDefinitions.html” from the “CodeExamples” folder in VS Code and then to its side open the FunctionDefinitionsSandbox.html” file:

With “FunctionDefinitions.html” open in VS Code, right click on “FunctionDefinitionsSandBox.html” in the “CODEESXAMPLES” folder and click on “Open to the Side.”



After the “FunctionDefinitionsSandbox.html” file is open, collapse the “Explorer” by clicking on the folders icon outlined in orange on the upper left-hand side of the screen. You VS Code window should now look as follows:

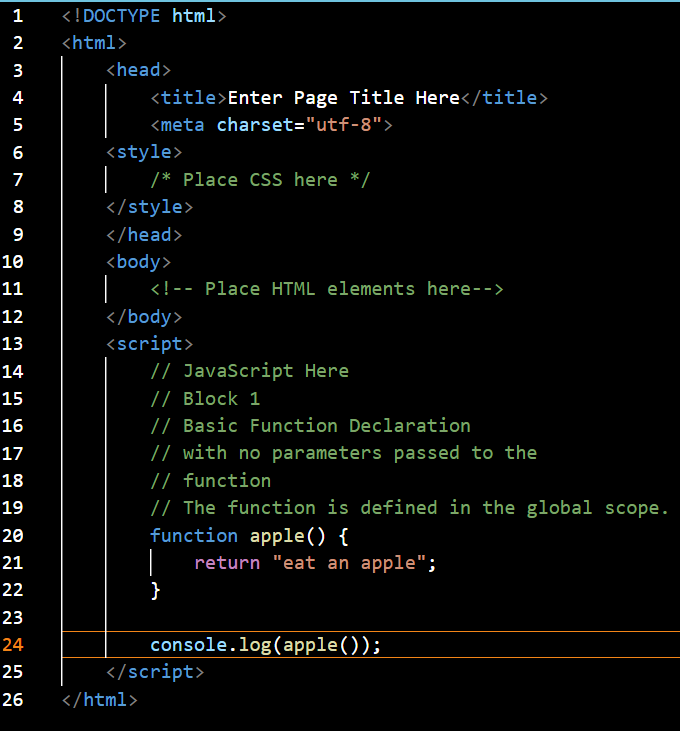
*Continued next page.*

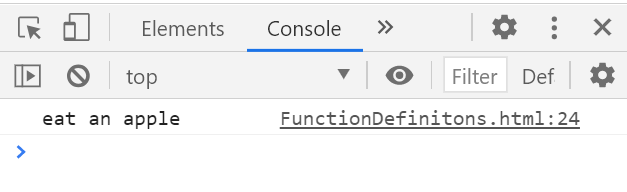


The “sandbox” code in the right panel contains blocks of code we will paste into the left panel.

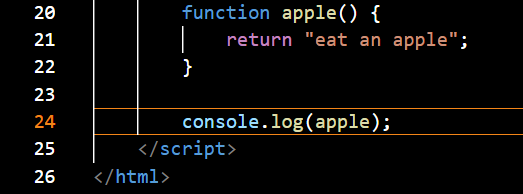
“Block 1,” in the “sandbox” is a basic function with no parameters passed to it but let us “play” with it a bit.

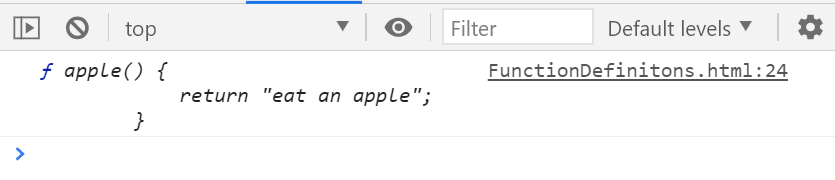
First paste “Block 1” into “FunctionDefinitions.html” and add a “console.log(apple())” after the function definition and run the code in the browser, open the programmer dev tools console in the browser.





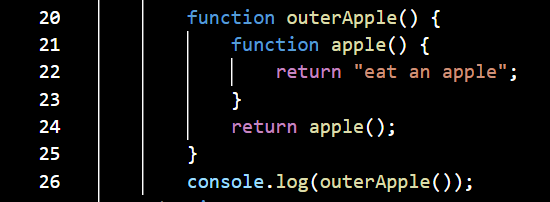
The result of the call to the function “apple” shown in the console is what we would expect. Now make a change the “console.log(…)” on line 24 to “console.log(apple)” and run the code in the browser again:

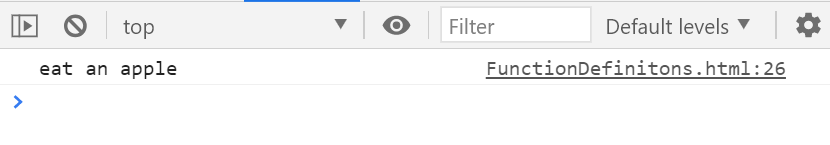




Notice that the output of the “console.log(…)” is the actual function definition of the function “apple().” The difference between the console.log(…) in the first example and the example above is that in the example above we left out the “()” following the function name “apple.” Without the “()”rather than invoke the “apple” function and write out the result of the invocation the console.log(…) will simply write out the definition of the function “apple.”

Now further modify the code as shown and run it in the browser.

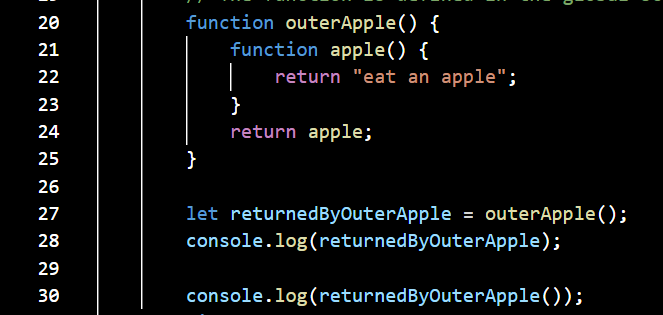




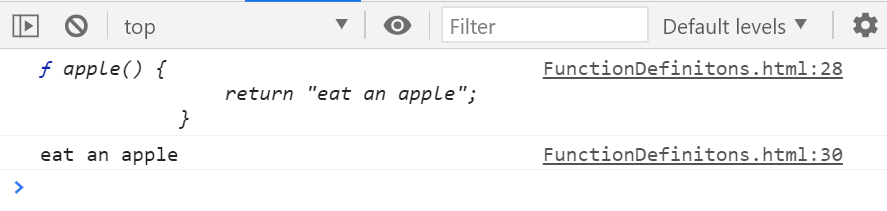
We now have the function “apple” defined inside the function “outerApple.” After we create the function “apple” we return **an invocation of it** – notice the “()” following “apple” on line 24. This means that when we call the function “outerApple” what is returned is an invocation of the function “apple” so that when we write the results of invoking “outerApple” to the console we see what is returned by the invocation of “apple.”

*Continued next page.*

Remove the “()” following apple on line 24, and add the code shown on lines 27 through 30, and run the code in the browser.

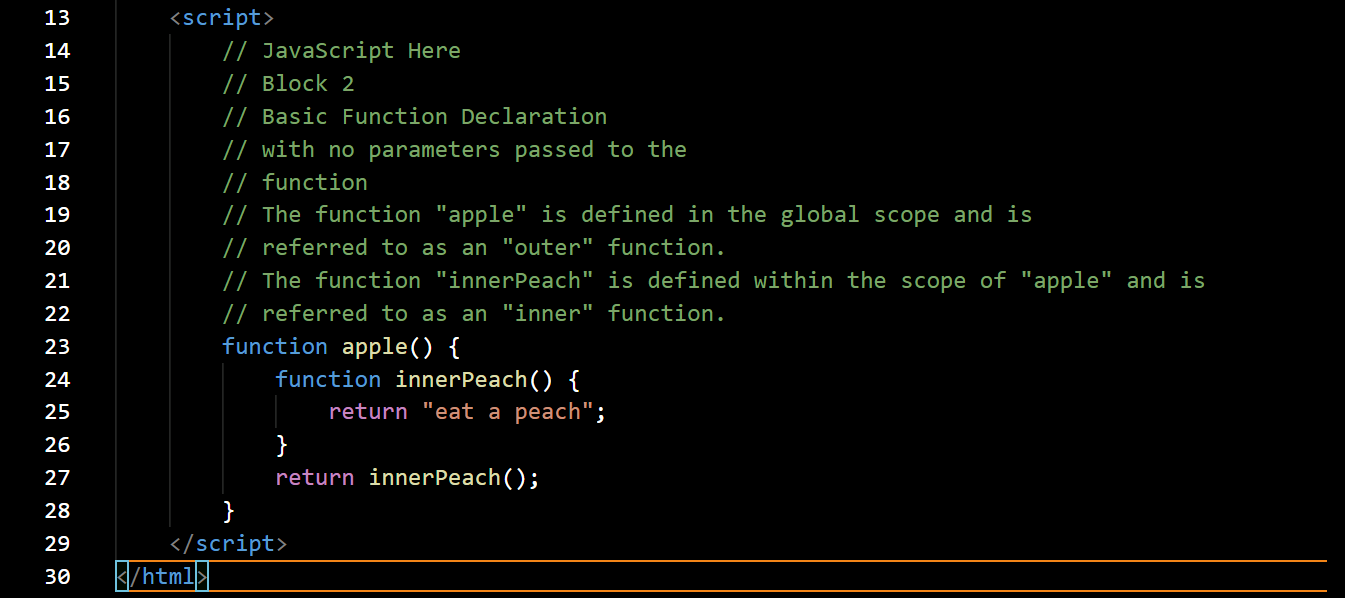


The function “outerApple” now returns the definition of the function “apple.” If we set the variable “returnedByOuterApple” to the results of invoking “outerApple”, as shown in the console we get the definition of the function “apple.” If we then write to the console the result of invoking “returnedByOuterApple,” we see the result of invoking “apple.”

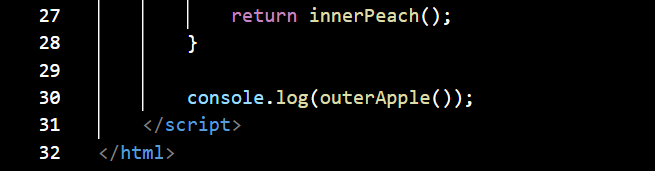


Keep all of this in mind as we move forward.

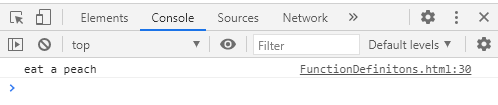
“Block 2,” shows a function definition, “function apple() {…”, which is defined in the “global” scope and is referred to, for this example, as an “outer” function which contains a function definition, “function innerPeach() {… , which is defined within the scope of “apple()” and is referred to as an “inner” function. Copy “Block 2” into “FunctionDefinitions.html” on the right-hand side:



Right before the </script> tab add a console.log() command as shown.



Save the changes and run the code in your browser. Make sure you have the “Console” tab in the “Developer Tools” window open. In the “Console” you will see:



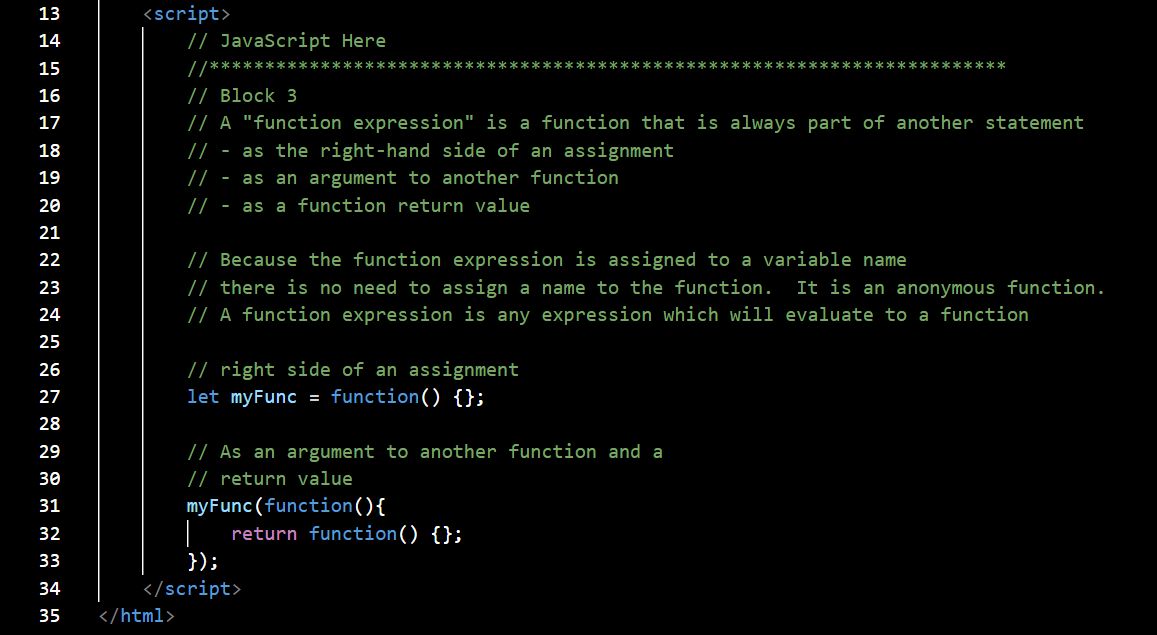
We invoked the function “apple()” within which is the function “innerPeach().” Following the definition of “innerPeach()” we placed a “return innerPeach()” statement which invoked the function “innerPeach” and then returned the results of calling “innerPeach” to our invocation of “apple()” on line 30. The return value from “innerPeach(),” is written out by the console.log() on line 30. This will become a familiar structure when we explore “callback” functions and function “closures.”

As an experiment, below the “console.log(outerApple());” statement enter the statement “console.log(innerPeach();” and examine the results.

The “sandbox” code in “Block 3,” is concerned with the definition of a “function expression.” A function expression is always part of another statement:

* As the right-hand side of an assignment
* As an argument to another function
* As a function return value

Copy the code from “Block 3,” into the “FunctionDefinitions.html” file:

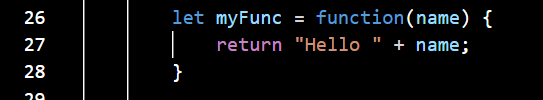


We are not going to execute anything here since the code here is “dummy code.”

On line 25, we show the variable “myFunc” is assigned the value of a function on the right-hand side of the equal sign. The function on the right-hand side is a “function expression.” To invoke it we would write “myFunc();”

On lines 29, through 31, we have the function “myFunc()” being invoked with another function as an argument, *“(function(){ return function() {}; });”* as well as returning a function on line 30. The function being passed as an argument and the function being returned are both function expressions. This will fall into place as we try out “real” examples shortly.

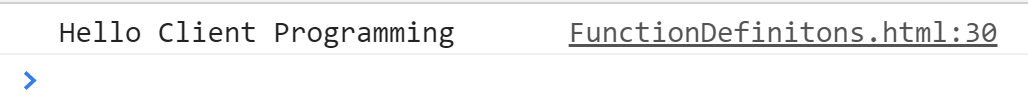
As an exercise, modify “myFunc” as shown below:



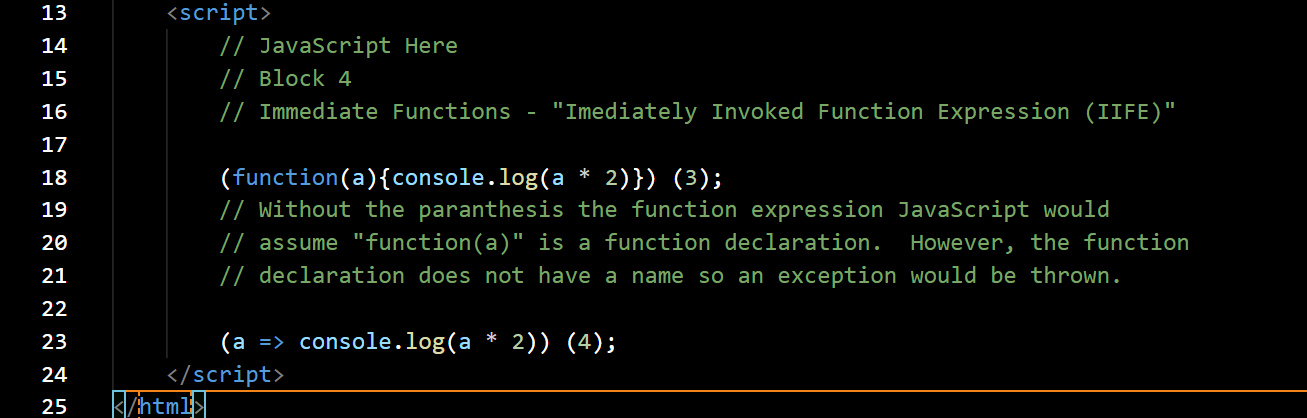
Then, add the line:



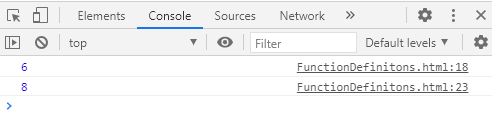
Comment out the remaining code in the <script> block and run the page.



“Block 4,” of the “sandbox” is concerned with function expressions known as “Immediately Invoked Function Expressions,” “IIFEs.” These are function expressions whose definition is immediately followed by an argument to be passed to the function and thus is immediately executed. Copy “Block 4” into “FunctionDefinitions.html.”



On line 18, we have a function defined which takes the parameter “a,” multiplies it by 2, and writes the result out using console.log(). You will notice that the function is enclosed in parenthesis, “().” *This tells JavaScript that what is inside the parenthesis is a function expression rather than a function definition. Immediately following the function expression is another pair of parentheses enclosing the integer 3. This is recognized by JavaScript as the input parameter for the function expression immediately preceding it* *and the function expression is immediately executed taking 3, as its input parameter.* On line 23, is the same function expression using “fat arrow” notation. Save the code and run it in your browser.



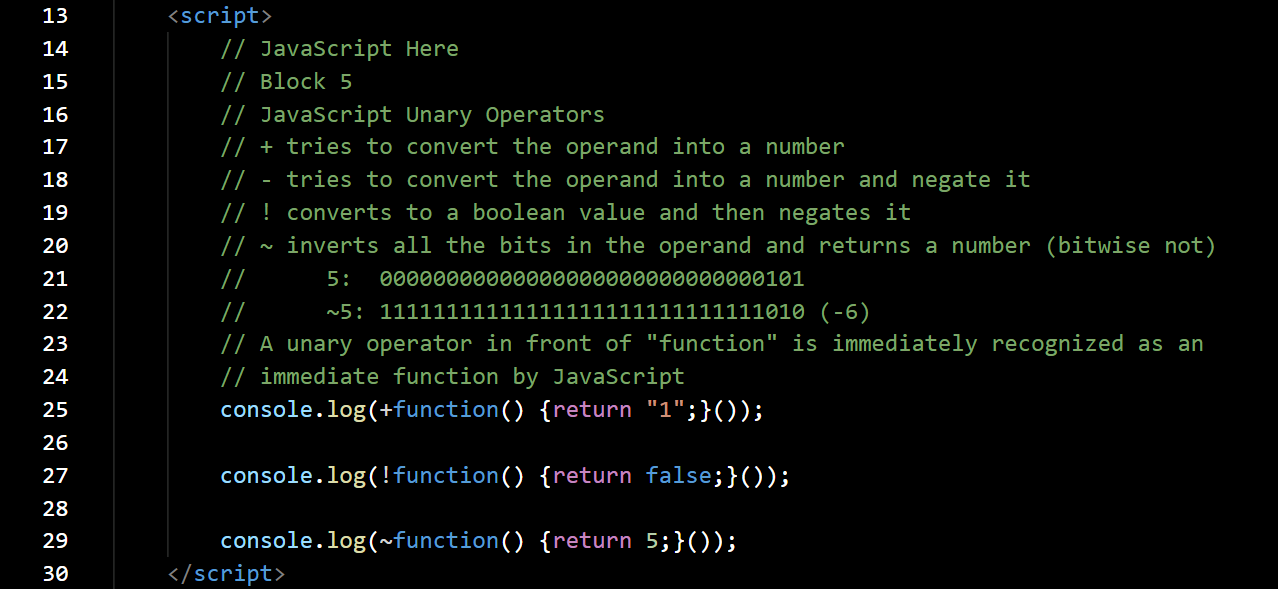
Note – IIFEs have application in the implementation of the “JavaScript Module Pattern,” a design pattern which was originally defined to provide both private and public encapsulation for classes in conventional software engineering.[[1]](#footnote-1) We’ll take a look at the “JavaScript Module Pattern” in a future lecture.

JavaScript unary operators can be used to create “IIFEs” by placing the unary operator immediately in front of a function expression. The unary operators this can be done with are:

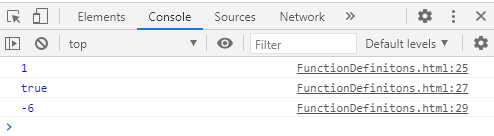
* + tries to convert the operand into a number
* - tries to convert the operand into a number and negate it
* ! converts to a Boolean and then negates it
* ~ inverts all the bits in the operand and returns a number (bitwise not)

When a unary operator is used in this way there is no need to enclose the function expression in parenthesis.

“Block 5,” of the “sandbox” has some examples of unary operator “IIFEs.” Copy them into “FunctionDefinitions.html.” Save your code and run it in the browser.



Save your code and run it in the browser. You will see the following output.



The output is self-explanatory.

See [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_NOT) for full explanation of “bitwise not.”

See [“How numbers are encoded in JavaScript”](https://2ality.com/2012/04/number-encoding.html) to understand how numbers are encoded in JavaScript: they are stored as floating-point numbers, same as a Java or C# double.

With some basic definitions out of the way and available for reference we will move on to callback functions.

**Callback Functions**

In the literature you will find two equivalent definitions for “callback:”

* “Whenever we set up a function to be called at a later time, whether by the browser in the event-handling phase or by other code, we’re setting up a callback. The term stems from the fact that we’re establishing a function that other code will later “call back” at an appropriate point of execution.[[2]](#footnote-2)
* A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.[[3]](#footnote-3)

Understanding callbacks can be a bit elusive at first so we will work through some examples. Before doing so I would like to spend a bit of time enhancing our JavaScript debugging tools.

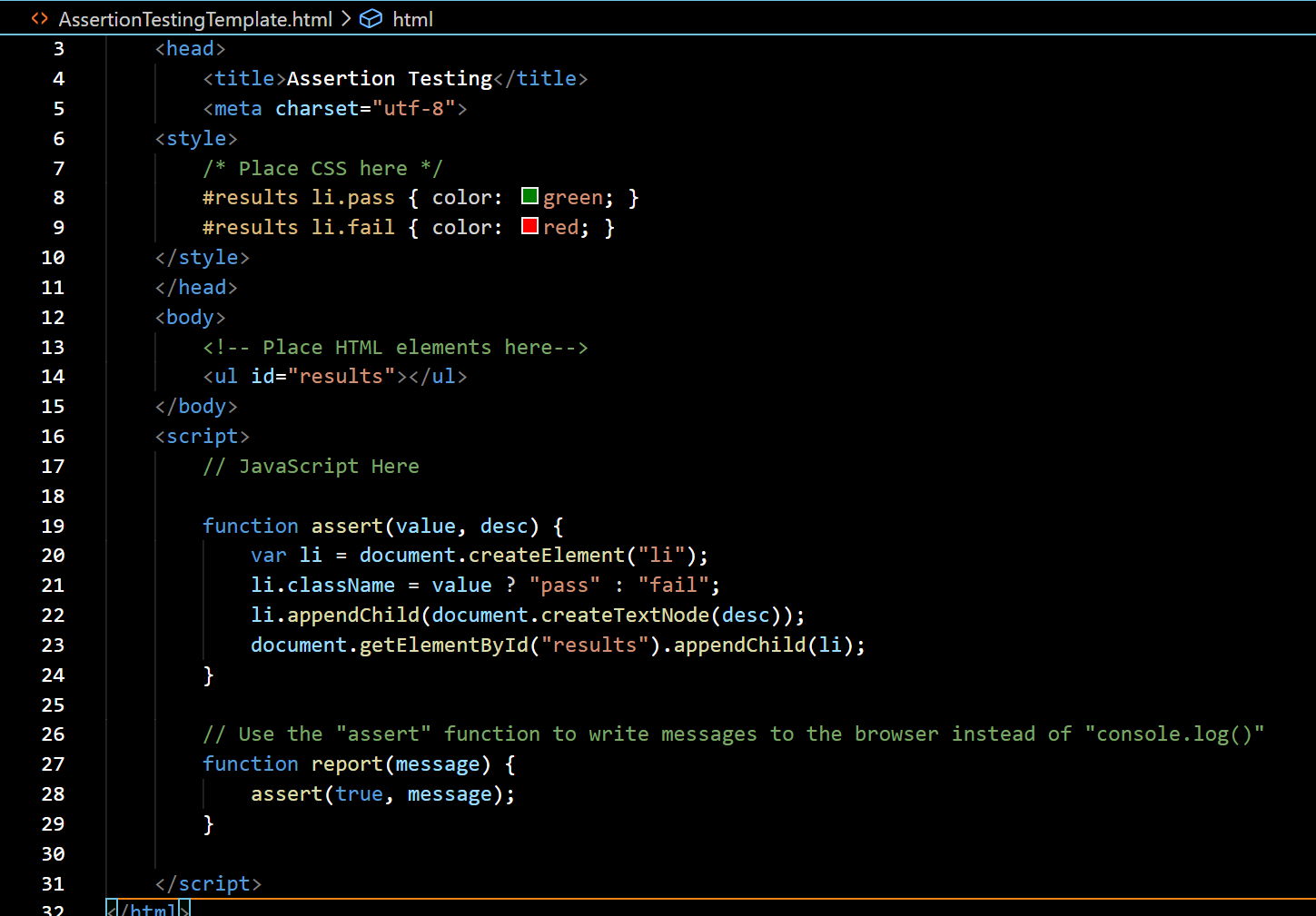
All of us have experience debugging the software we write, or if you’ve worked in the “real” world,” more than likely you’ve had to debug someone else’s software. There are all sorts of approaches to debugging with some being more formal than others. One method used by most of the established software testing tools is “assertion,” e.g., “xunit” in “Microsoft Visual Studio.”

“An assertion is a boolean expression at a specific point in a program which will be true unless there is a bug in the program. *A test assertion is defined as an expression, which encapsulates some testable logic specified about a target under test.”[[4]](#footnote-4)*

In “appendix B: Arming with testing and debugging” in Secrets of the JavaScript Ninja a handy “assertion” tool is provided which we will start using in our work. Among its nice features is that there is no longer the necessity to use “console.log()” to inspect what is happening in our code; this also removes the issue of “catching” the console.log() calls at the right time using breakpoints.

From the “CodeExamples” folder open the file “AssertionTestingTemplate.html” in VS Code.

*Continued next page*



There are three sections of our “html” page that we need to add code to in order to use the “assertion testing” tool:

* <style>
* <body>
* <script>

In the <style> section on lines 8, and 9, you will see two styles for the class name “results:”

* The first will assign the color “green” to a “list item” element with the “className” of “pass.”
* The second will assign the color “red” to a “list item” element with the “className” of “fail.”

In the “<body> is an empty “unordered” list, <ul> with the id=“results.”

Finally, in the script block are two functions:

* assert()
* report()

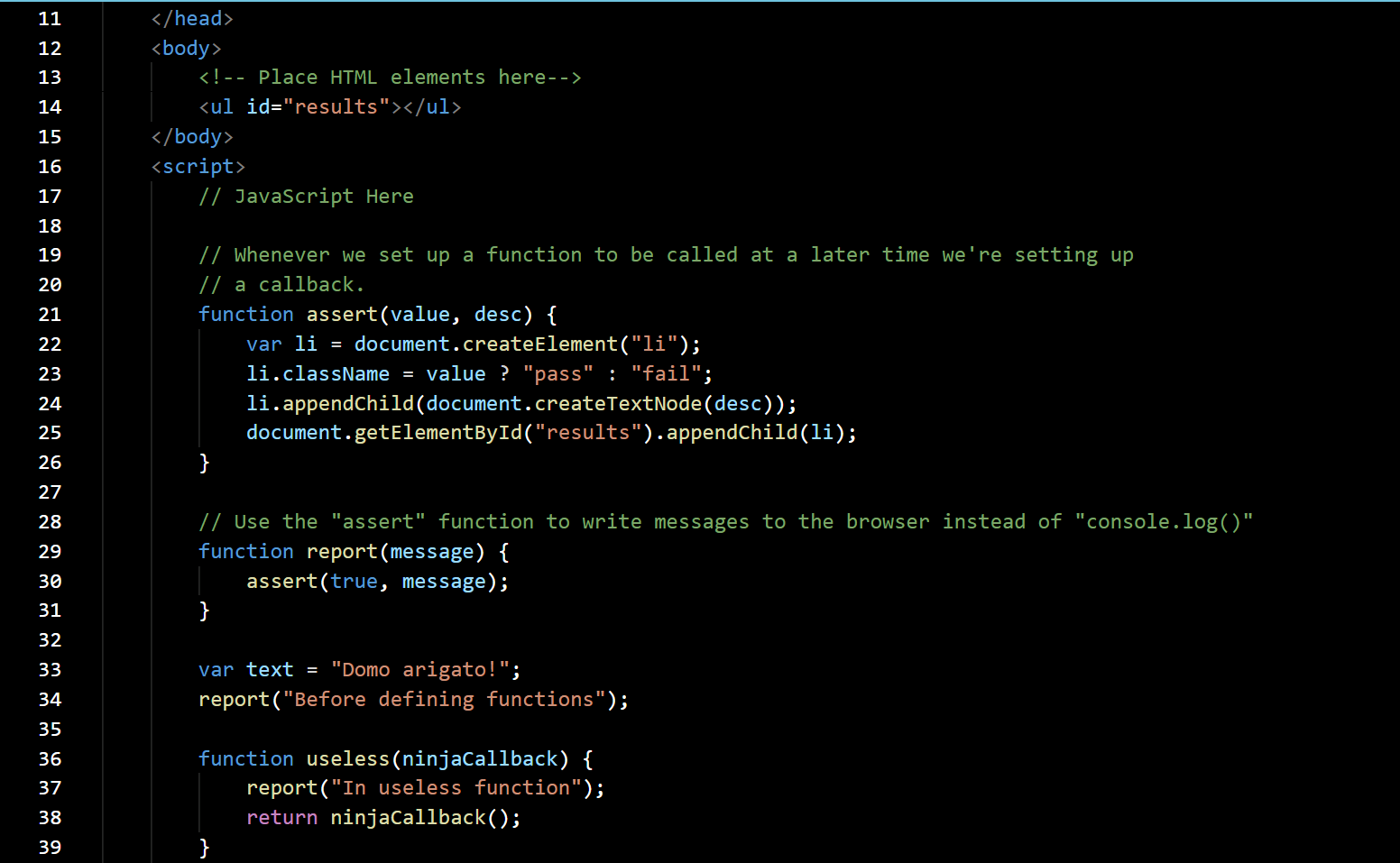
The “assert()” function is passed a “value” and a “description” where:

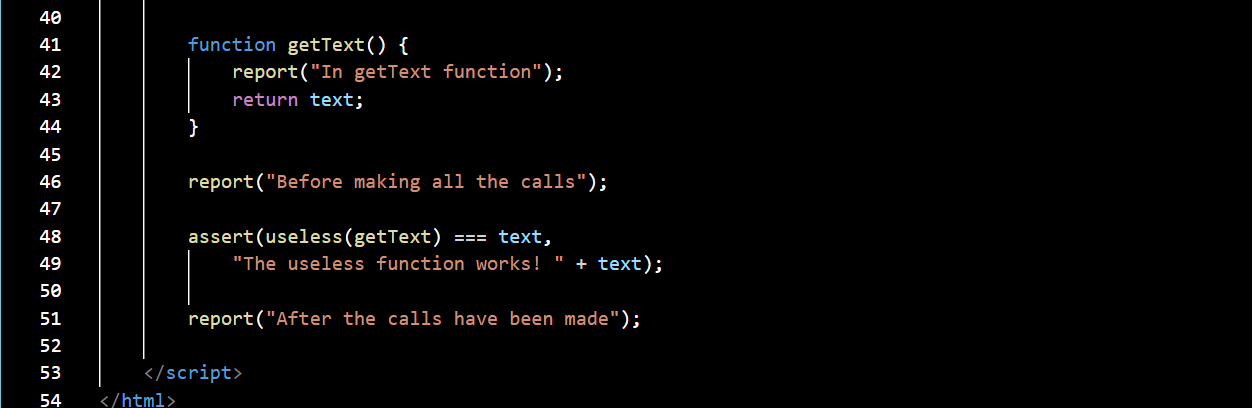
* value =
  + true if the “assertion” we’re testing “passed”
  + false if the “assertion” we’re testing “failed”
* description – the text that is written to the “unordered” list by the “assertion” test.

**The “report()” function calls the “assert()” function with “value=true” and the text of the message it wants written to the “unordered” list. It is simply a “reporting” tool.”**

We will try out the “assertion” test with a simple demonstration of the basic “callback” mechanism, this example is from “Secrets of the JavaScript Ninja,” pp 37.

Open the file “UselessCallback.html” from the “CodeExamples” folder in VS Code.





A flurry of code due to the inclusion of our “assertion testing” but things are easily sorted out.

On line 33, we have a variable “text” assigned the string “Domo arigato!,” “Thank you very much.”

Keeping in mind that as the web page loads the JavaScript script block is executed from “top to bottom” as soon as it is encountered. The first function call that will be executed is on line 34. This will post a message that indicates we are about to get started defining our functions.

The first function is “useless(),” it is just there to put in place the minimal amount of code to demonstrate a callback. “useless” takes a parameter “ninjaCallback.” Inside of “useless()” is a call to the report function to write the message “In useless function.” It will execute when “useless()” is called and show us the progress of our program.

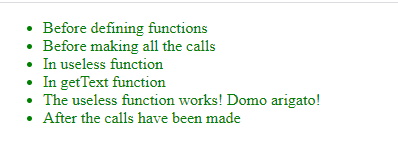
The next function we define is “getText()” which takes no parameters. It simply calls the report function to write the message “In getText function.” It will execute when “getText()” is called and show us the progress of our program as well as return the “var” text.

On line 46, we write out the message that we are at the point in the program “Before making all the calls” to the functions we defined.

Line 48, is where we have the “assert” function invoke “useless(getText) and compare its return value and type to the text “Domo arigato!.” If the assertion is true, the text “The useless function works!” will appear in green, if it fails, it will appear in red.

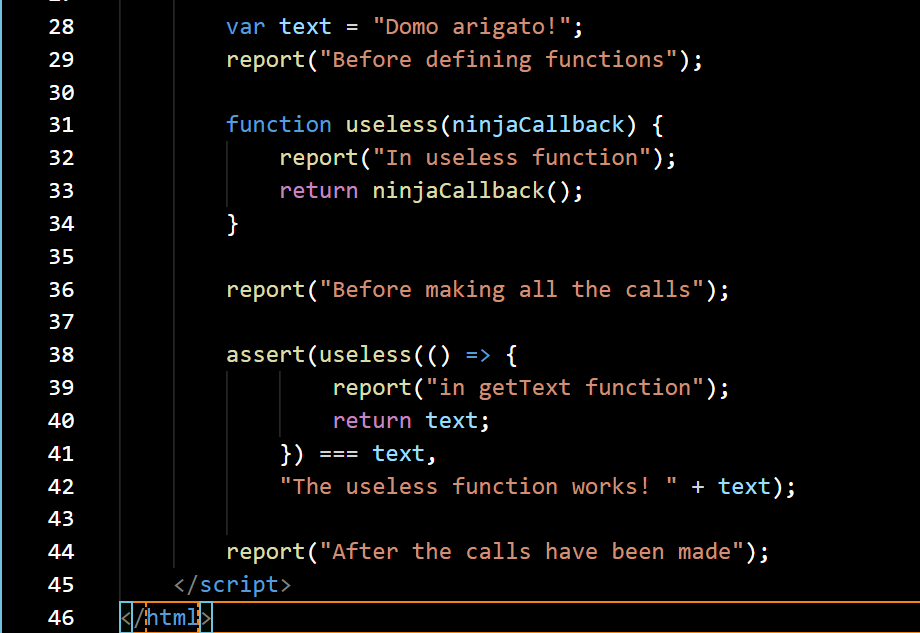
If you go back to our basic definitions for functions, you will remember that *“a callback function is a function passed into another function as an argument,* which is then invoked inside the outer function to complete some kind of routine or action.” According to that definition the function “getText()” is a callback function since it is passed as an argument to the “useless()” function.

Let us run the “UselessCallback.html” file in the browser. The output we get is, notice no need to bother with the “Console.”



* On line 48, we invoke “useless(getText).”
* The message on the third line of the unordered list is generated on line 37, of the “useless” function and tells us that upon that invocation we have entered the function.
* After that message is written we then execute the code on line 38, “return ninjaCallback()” is executed.
* This results in a call to “getText()” which on line 42, inserts in the unordered list, “In getText function.”
* On line 43, the contents of the “text” variable is returned to “useless().”
* Now back on line 38, of “useless(), the value of “text” is returned to line 48, of the program where it was invoked.
* The return value is then passed to the “assert()” function. The equivalence test “useless(getText) === text” returns “true” so the assert function writes out “The useless functionworks…” message to the unordered list in green.
* The last message in the unordered list indicates the end of the program.

The file “UselessCallback2.html” has a slightly modified version of the above program. Open it in VS Code. Below I am showing only the code that follows the definitions of the “assert()” and “report()” functions.



On line 31, you can see our “useless()” function. However, there is no longer a definition for the “getText()” function.

If you look at the “assert()” call on line 38, you will see that rather than passing the “getText()” function as an argument to the “useless()” function we are defining a callback function directly as the argument:

**() => {   
 report("in getText function");  
 return text;  
 }**

***We can do this because JavaScript gives us the ability to create functions in the code anywhere an expression can appear.***

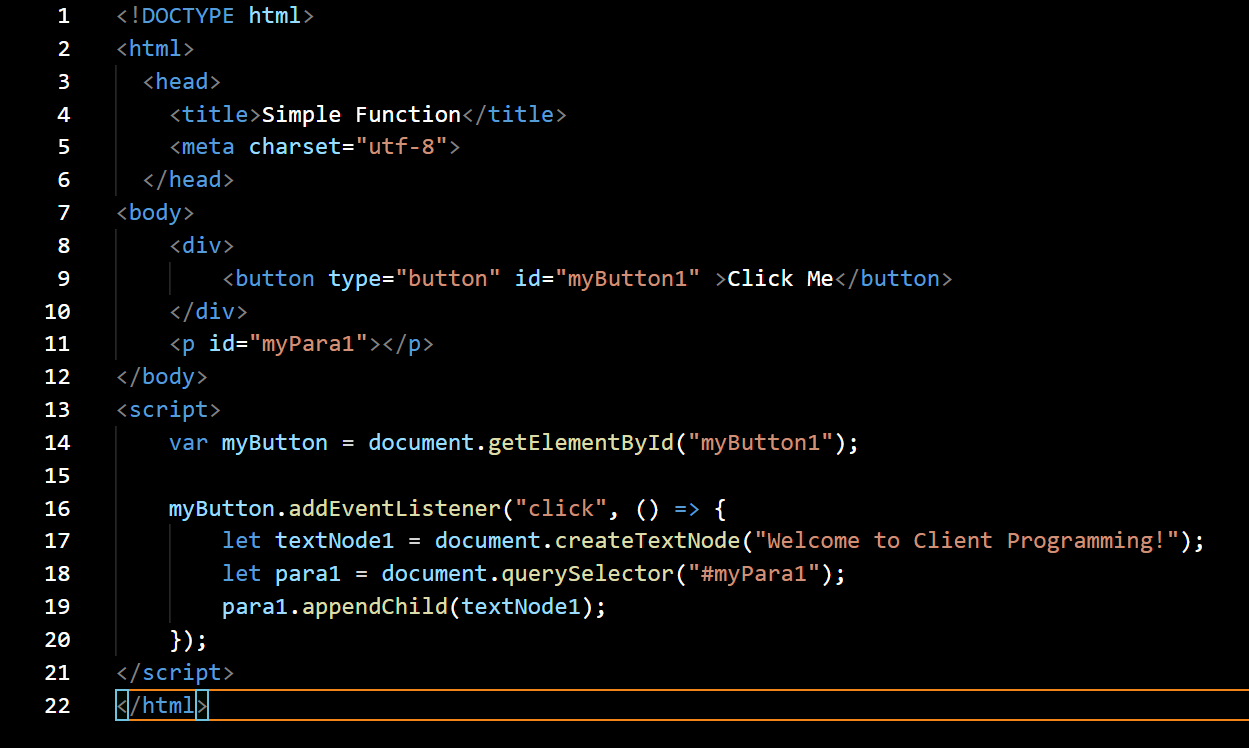
There are two additional examples of this type of callback function usage in the “CodeExamples” folder:

* NameCallback.html
* NameCallback2.html

I leave it to you to explore and run these two examples.

In the above examples we have been calling “our own” callback. However, if you remember back to creation of an “EventListener” this was a situation where the browser called a function based on an event, e.g. a mouse click.

Open “DOMEventTargetInterface.html” from the “CodeExamples” folder.



On line 16, we call the “EventTarget,” “myButton” is the event target, method “addEventListener().” “addEventListener()” takes two arguments:

* An event, in this case the “click” event
* A function to be executed upon detection of the event by the browser.

What is important here is that the second argument being passed to the “.addEventListener()” method is a function. A function passed as an argument to another function is a “callback”, so the function executed upon detection of the “click” event is a callback function.

In fact, we are doing here what we did above in the file “UselessCallback2.html,” we are defining a callback function directly as the argument.

Final note, at least for the moment, on “callbacks.” In our “UselessCallback” we called our own “callback.” In the “addEventListener” example above the web browser called the “callback” on the “click” event. When we call the “callback” from our own code, the “callback” is “synchronous.” When a mechanism external to our code, e.g. the browser, calls the “callback,” the “callback” is asynchronous.

We will soon be working with “HTTP” requests / responses. When an “HTTP” request is made to a server, a “callback” function is created to be called by the “browser” when the “browser’s” even loop detects the “HTTP” response – this is an asynchronous “callback.”

**Is JavaScript Compiled?**

We “traditionally” think about JavaScript as an “interpreted” language by which we mean that as the JavaScript code is loaded into the browser the browser starts executing it. That is not “how it happens.”

The processing of JavaScript programs occurs in, at least, two phases:

* Parsing / compilation
* Execution

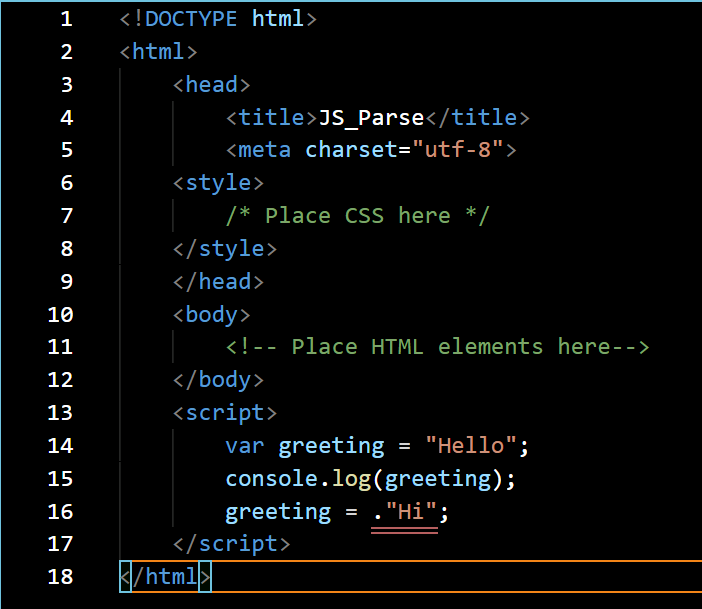
The separation of a parsing/compilation phase from the subsequent execution phase is observable fact, not theory or opinion. While the JS specification does not require “compilation” explicitly, it requires behavior that is essentially only practical with a compile-then-execute approach.[[5]](#footnote-5)

There are three program characteristics you can observe to prove this:

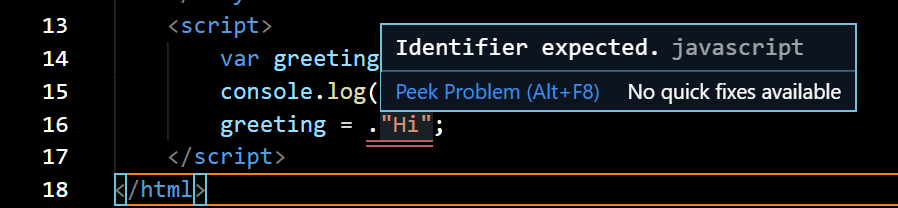
* Syntax errors
* Early errors
* Hoisting errors

We will explore the details of “hoisting” and “early errors” in a future lecture. For now, we will look at “syntax errors” and a simple “hoisting error” to show that they are detected during the “parsing / compilation” phase rather than the execution phase.

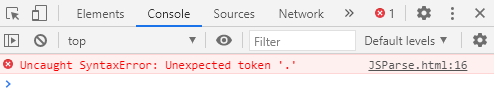
Open the file “JSParse.html” from the “CodeExamples” folder in VS Code.



The double orange line under “.Hi”; on line 16, is an Identifier expected error detected by a VS Code language specific feature, not anything to do with how the code will run in the browser.



Let us run the code “as is” in the browser. This is the output we see in the “Console” tab:

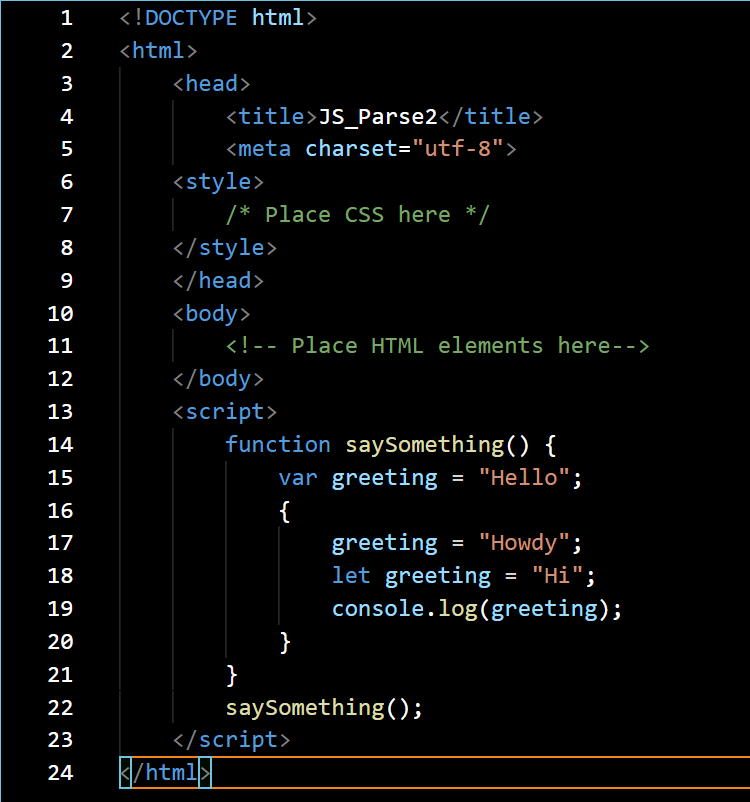


No output was produced by the program. Rather, a “SyntaxError” is thrown. If you look at the code, you will see that the line on which the “SyntaxError” is thrown, line 16, happens after the “well formed” “console.log(greeting);” on line 15. If JavaScript was being “interpreted” line by line from the top down, we should see the output of the “console.log(greeting)” call followed by the “SyntaxError” being thrown which did not happen.

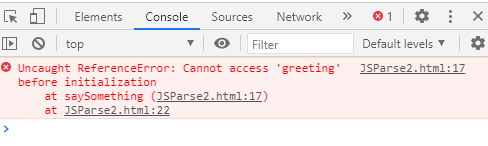
The only way the JavaScript engine could know about the “SyntaxError” on line 16, before executing the code on lines 14, and 15, is by the JavaScript engine first parsing the entire program before any of it is executed.

Let us look at a different example of the result of “compiling before execution.” Open the file “JSParse2.html” in VS Code:

*Continued next page.*



VS Code has not “flagged” any errors so let us run it in the browser without further consideration. When it runs, we get the following output in the “Console” tab.

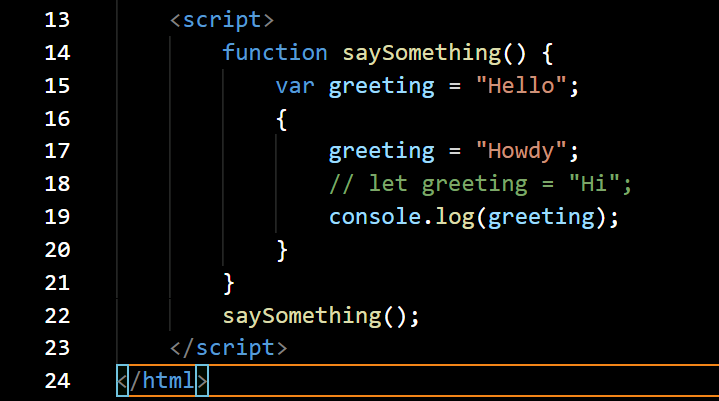


So, this looks like a strange error. We have the variable “greeting” defined as the first line of “saySomething(),” line 15. Inside the “block” extending from line 16, to line20, we reassign the value of greeting to “Howdy,” line 17, and on line 19, write it out to the console with a “console.log().”

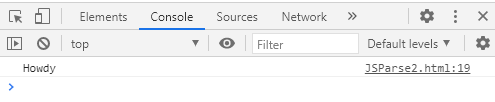
What is causing this error, and something we did not mention in the previous paragraph, is the ‘let greeting = “Hi”;’ statement on line 18. The variable “greeting” defined with the “var” on line 15, is in scope throughout the function “saySomething().” That is, any line of code within the function can reference the “greeting” definition on line 15, and get the value “Hello,” well almost any line of code.

As mentioned, we have a block of code defined from lines 16, through 20, delimited with opening and closing curly braces, { }. Within that block of code we have another variable “greeting” defined with the statement ‘let greeting = “Hi”;’ The scope of a “let,” as well as a “const,” is the “closest enclosing block.” In other words, within the block the variable “greeting” defined with the “let” is in scope and within that block referencing the variable “greeting” will return the value “Hi.” The “ReferenceError” shown on line 17, comes about because the code attempted to reference the variable “greeting” in the block but within the block the variable “greeting” is not defined until line 18. The error could only be produced if the JavaScript engine “parsed” the code prior to execution and mapped out the scope of all the variables, otherwise it would have happily used the variable “greeting” from line 15.

If we comment out line 18, in the above code:



Save it and run it in the browser again, we see:



This time the console.log() worked. The only definition of the variable “greeting” is the one on line 15, and that definition can be accessed anywhere within the function. Within the code block delimited by the curly braces there is a reference to greeting on line 17, where its value is being reassigned to “Howdy.” When “parsing” the code the JavaScript Engine could not find a variable definition for greeting in scope within the code block, so it looked to the next higher scope, the function, for the definition. It found the definition there so within the block it was able to use that definition to bind “greeting” to the value “Howdy” and everything worked.

*Creating scopes based on the position of variables, functions, and blocks within the JavaScript code during the “compile” phase is known as “lexical scoping.” Simply put, scope in a JavaScript program is determined at “compile time.”* The lexical scope can be defined as a set of rules and boundaries defined for variables in the program, while the program is compiled. These rules are static, they can only be referred to but not be changed while executing the program.

This not so brief introduction to “scopes” will be useful when we examine “closures.”

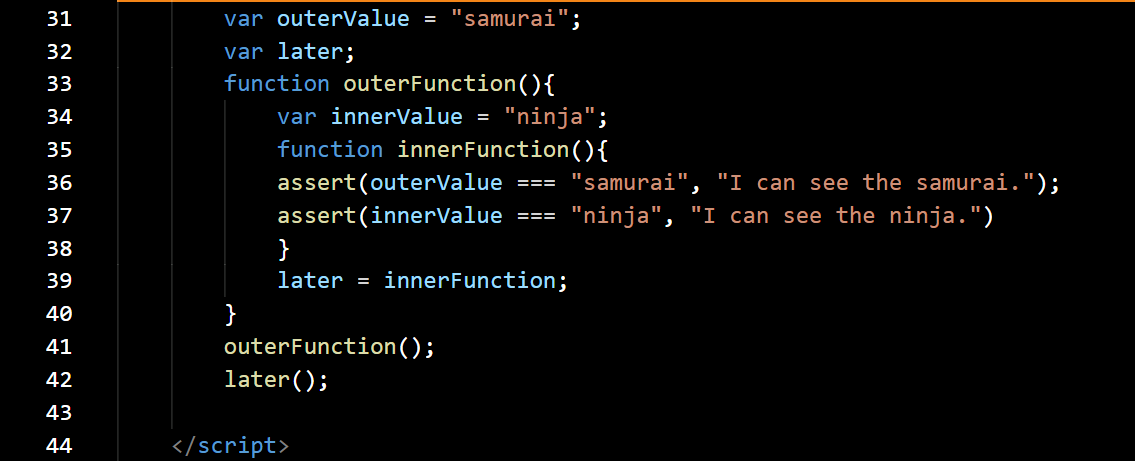
**Closures[[6]](#footnote-6)**

A closure allows a function to access and manipulate variables that are external to the

function. ***Closures allow a function to access all the variables, as well as other functions,***

***that are in scope when the function itself is defined.***

Open the file “NinjaClosure.html” from the “CodeExamples” folder in VS Code. We are using the “assert” debugging environment here so I’m only showing the code in the <script> block that is specific to demonstrating a “closure.”



On lines 31, and 32, we have two variables defined in the “Global” scope. The first, “outerValue” is assigned the string “samurai.” The second, “later,” is defined but not assigned a value.

We then create the function “outerFunction()” within which, on line 34, we define the variable “innerValue” and set it to the string “ninja.” As we saw earlier when discussing “scope,” the variable “innerValue” is accessible only within the function scope, i.e. within “outerFunction().”

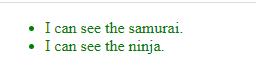
We then create the function “innerFunction()” within “outerFunction().” Because JavaScript uses “Lexcial Scoping” during the “compile” phase of the JavaScript engine the variable “innerValue” is in scope for “innerFunction()” when “innerFunction()” is created.

On line 39, we set the value of the variable “later” to “innerFunction.” Because the variable “later” is in the “global” scope we will be able to call “innerFunction” at any time.

On line 41, we call “outerFunction()” which results in “innerFunction()” being created and its reference assigned to “later.” However, there is no call to “innerFunction” made at this point.

On line 42, we call “innerFunction()” through “later.” We cannot call “innerFunction()” directly since its scope is limited to within “outerFunction.”

When we run this code in the browser we see:



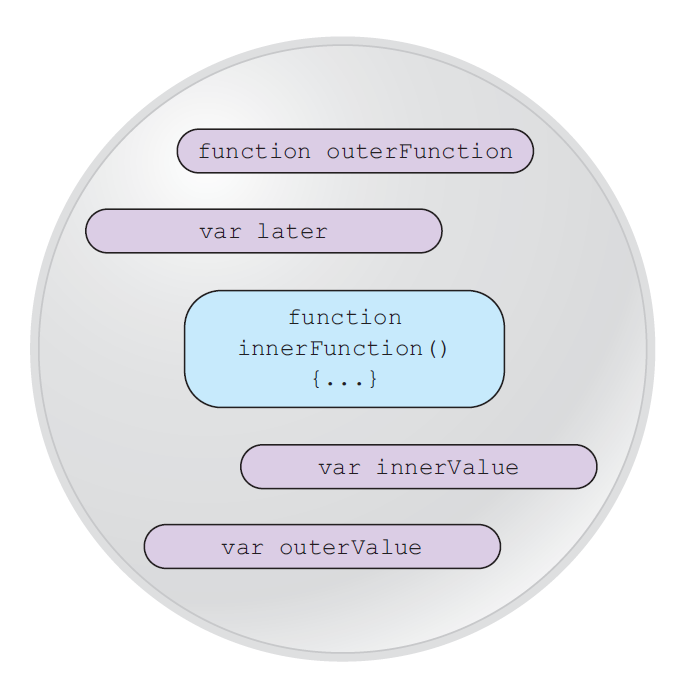
It is important to keep in mind that once a function finishes executing, its scope is lost.

When we call “innerFunction()” the function “outerFunction()” has already run and finished and it is “out of scope.”

When we then call “innerFunction()” the first line of output is created because the assertion: outerValue === “samurai” is “true.” That is expected because “outerValue” is in the global scope, so it is always accessible.

However, the assertion on the second line of “innerFunction()” is also true which means that the variable “innerValue” which is part of the scope of “outerFunction” is somehow still available.

The reason “innerValue” is still available to “innerFunction” is that when we declare “innerFunction” inside “outerFunction,” not only is the function declaration defined, but a closure is created that encompasses the function definition as well as all variables in scope at the point, i.e. time, of function definition. That is what a “closure” is. A “closure” creates a “safety bubble” of the function variables in scope at the point of the function’s definition so that when the function runs it has all that it needs to execute. The picture below is a conceptual drawing of the “bubble.”



1. [Learning JavaScript Design Patterns](https://addyosmani.com/resources/essentialjsdesignpatterns/book/), Addy Osbmani, ©Addy Osmani 2012-2020, available under a “Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 unported license. [↑](#footnote-ref-1)
2. Secrets of the JavaScript Ninja, pp. 36 [↑](#footnote-ref-2)
3. [MDN web docs](https://developer.mozilla.org/en-US/docs/Glossary/Callback_function) [↑](#footnote-ref-3)
4. [Assertion Testing](https://www.tutorialspoint.com/software_testing_dictionary/assertion_testing.htm#:~:text=An%20assertion%20is%20a%20boolean,about%20a%20target%20under%20test.) [↑](#footnote-ref-4)
5. You Don’t Know JS Yet – Scope & Closures, Second Edition, Kyle Simpson, © 2020 Getify Solutions Inc. [↑](#footnote-ref-5)
6. Material in this section is from “Secrets of the JavaScript Ninja,” pp. 93, 94 [↑](#footnote-ref-6)