Functions: Continued

- Due No Due Date
- Points 1
- Submitting a website url

 (https://github.com/learn-co-curriculum/phase-1-javascript-functions-continued)  (https://github.com/learn-co-curriculum/phase-1-javascript-functions-continued/issues/new)

Learning Goals

- Define a function using a function declaration
- Define `hoisting`
- Define `function expression`
- Define `anonymous function`
- Define a function using a function expression
- Define an IIFE: `Immediately-Invoked Function Expression`
- Define `function-level scope`
- Define `scope chain`
- Define `closure`

Introduction

This lab describes some more advanced concepts related to JavaScript functions. Be sure to take time to experiment or read up on a concept if you're not comfortable with the idea before moving on. If you're struggling here, the remainder of this module will be challenging. Fix any gaps now before moving on.

We also recommend that you complete the lab as you read through the sections. Reinforcing what you read by physically typing in the code will help make sure the concepts are locked in. We'll prompt you when it's a good time to shift modes from "reading along" to coding.

Getting Started

If you haven't already, fork and clone this lab into your local environment. Remember to **fork** a copy into your GitHub account first, then **clone** from that copy. Navigate into its directory in the terminal, then run `code .` to open the files in Visual Studio Code.

Define a Function Using Function Declaration

In JavaScript, the most common way to define functions is with a **function declaration**:

```
function razzle() {
  console.log("You've been razzled!");
}
```

The word `razzle` becomes a *pointer* to some stored, potential, not-yet-actually-run bit of work (the function). We use the *pointer* to *call* or *invoke* the function. We *call* the function by adding `()` after the *pointer*.

```
function razzle() {
  console.log("You've been razzled!");
}
razzle();
//=> "You've been razzled!"
```

Interestingly, you can write function declarations *after* you call them:

```
razzle(); //=> "You've been razzled!"
function razzle() {
  console.log("You've been razzled!");
}
```

Functions can be passed arguments, given default arguments, etc. Here's a brief code synopsis:

```
function razzle(lawyer = "Billy", target = "'em") {
  console.log(`${lawyer} razzle-dazzles ${target}!`);
}
```

```
razzle(); //=> Billy razzle-dazzles 'em!
razzle("Methuselah", "T'challah"); //=> Methuselah razzle-dazzles T'challah!
```

LAB: Implement a function called `saturdayFun`. It should return a `String` like `"This Saturday, I want to ....!"` Fill in the `...` with the activity that's passed in as the first parameter. If nothing is passed in, default to `"roller-skate"`. Run `npm test` to verify you've gotten the first set of tests passing before continuing with the lesson.

Define `Hoisting`

JavaScript's ability to call functions *before* they appear in the code is called *hoisting*. For hoisting to work, **the function must be defined using a function declaration**.

Define `Function Expression`

We've learned that programming languages feature *expressions*: arrangements of constants, variables, and symbols that, when interpreted by the language, produce a *value*. To review, open up your browser console and type in these examples:

```
1 + 1; //=> 2
"Razzle " + "dazzle!"; //=> "Razzle dazzle!"
```

The examples above are expressions that return *primitive values*, but JavaScript also has *function expressions* that look like this:

```
function() {
  console.log("Yet more razzling");
}
```

The *value* returned by this expression is the function itself. Go ahead and enter the above into the browser console; you should see the following:

```
Uncaught SyntaxError: Function statements require a function name
```

The problem is that, when the function expression appears by itself as shown above, **JavaScript does not recognize it as a function expression**; it instead interprets it as a function declaration that's missing its name. One way to tell the JavaScript engine that it's a function expression is to use

the `grouping operator ()` to wrap the entire thing:

```
(function () {
  console.log("Yet more razzling");
});
```

Recall that the grouping operator is usually used in arithmetic operations to tell the JavaScript engine to evaluate the value that's inside it first. It's serving a similar purpose in this case: it's telling JavaScript to interpret what's inside the parentheses as a *value*. With the grouping operator in place, the JavaScript engine recognizes our function as a function expression. Enter the function into your console again, this time using the grouping operator. You should see the following:

```
ƒ () {
  console.log("Yet more razzling");
}
```

JavaScript now correctly shows us the return value of our function expression: a *function* (indicated by the `ƒ ()`) storing the work of logging our message.

Define `Anonymous Function`

An **anonymous function** is, quite simply, a function that doesn't have a name:

```
function() {
  console.log("Yet more razzling");
}
```

Unlike a function declaration, there's no function name in front of the `()`. Note, however, that if we don't assign a name to the function, we have no way to call it. We lose access to our function immediately after it's created. So how can we invoke an anonymous function? We've seen one way before: we can use it as a callback function. For example, you'll often see anonymous functions passed as an argument to an event listener:

```
const button = document.getElementById("button");
button.addEventListener("click", function () {
```

```
  console.log("Yet more razzling");
});
```

Our anonymous function is being passed as an argument to `addEventListener`. The JavaScript engine "stores it away" as work to be executed later, when the button is clicked.

Define a Function Using a Function Expression

Another way we can solve the problem of accessing an anonymous function is by declaring a variable and assigning the function as its value. Recall that any expression can be assigned to a variable; this includes function expressions:

```
const fn = function () {
  console.log("Yet more razzling");
};
```

The code above defines our function using a function expression. If we ask JavaScript what's in `fn`, it tells us:

```
fn; //=> ƒ () { console.log("Yet more razzling") }
```

Here, `fn` is a *pointer* to the stored block of work that hasn't yet been invoked. Just as with **function declaration**, to actually do the work, we need to *invoke* or *call* the function. We do this by adding `()` to the end of our "pointer", the variable name:

```
const fn = function () {
  console.log("Yet more razzling");
}; //=> undefined
fn; //=> ƒ () { console.log("Yet more razzling") }
fn(); // "Yet more razzling"
```

Also as with a function declaration, if we need to pass arguments to the function, we would include those in the parentheses when we call the function.

We now know how to define a function as a function expression. Very importantly, ***function expressions are not hoisted***. The same is true for any variable assignment: if we assign a `String` or the result of an arithmetic expression to a variable, those assignments are not hoisted either.

LAB: Implement a function expression called `mondayWork`. The function should return a `String` like `"This Monday, I will ... ."` Fill in the `...` with the activity that's passed in as the first parameter. If nothing is passed in, default to `"go to the office"`. Run `npm test` to verify you've gotten this set of tests passing before continuing with the lesson.

Define an IIFE: Immediately-Invoked Function Expression

Another way to invoke an anonymous function is by creating what's known as an `immediately-invoked function expression (IIFE)`.

As a thought experiment, consider what happens here:

```
(function (baseNumber) {
  return baseNumber + 3;
})(2); //=> ???
```

We recognize the first `()` as the grouping operator that tells the JavaScript engine to interpret the contents as a value — in this case, a function expression. What this means is that, in the IIFE statement, the value returned by the first set of parentheses is an anonymous function, which can be invoked (immediately).

The second `()` are the `()` of function invocation. When we put them immediately after the first set of parentheses, we're invoking the function that those parentheses return immediately after defining it. Try it out in the browser console:

```
(function (baseNumber) {
  return baseNumber + 3;
})(2); //=> 5
```

Interestingly, any variables, functions, `Array`s, etc. that are defined *inside* of the function expression's body *can't* be seen *outside* of the IIFE. To see this, check the value of `baseNumber` in the console. It's like opening up a micro-dimension, a bubble-universe, doing all the work you could ever want to do there, and then closing the space-time rift. We'll see some of the practical power of "hiding things" in IIFEs a little later in this lesson.

Define `Function-Level Scope`

JavaScript exhibits "function-level" scope. This means that if a function is defined *inside another* function, the inner function has access to all the parameters of, as well as any variables defined in, the outer function. This works recursively: if we nest a third function inside the inner function, it will have access to all the variables of both the inner and outer enclosing functions. Each of the enclosing parents' scopes are made available via the *scope chain*. We will define the scope chain a bit later in this lesson. Let's start by seeing it in action.

> **ASIDE**: This is where people **really** start to get awed by JavaScript.

Consider this code:

```
function outer(greeting, msg = "It's a fine day to learn") {
  // 2
  const innerFunction = function (name, lang = "Python") {
    // 3
    return `${greeting}, ${name}! ${msg} ${lang}`; // 4
  };
  return innerFunction("student", "JavaScript"); // 5
}

outer("Hello"); // 1
//=> "Hello, student! It's a fine day to learn JavaScript"
```

Let's break this down:

1. We call `outer`, passing `"Hello"` as an argument.
2. The argument (`"Hello"`) is saved in `outer`'s `greeting` parameter. The other parameter, `msg`, is set to a default value.
3. Here's our old friend the function expression. It expects two arguments, to be stored in the parameters `name` and `lang`, and `lang` is assigned the default value of `"Python"`. The function expression itself is saved in the local variable `innerFunction`.
4. Inside `innerFunction` we make use of its parameters, `name` and `lang`, ***as well as*** the `greeting` and `msg` parameters defined in innerFunction's containing (parent) function, `outer`. `innerFunction` has access to those variables via the scope chain.
5. Finally, inside `outer`, we invoke `innerFunction`, passing arguments that get stored in `innerFunction`'s `name` and `lang` parameters.

This might look a little bit weird, but it generally makes sense to our intuition about scopes: inner things can see their parent outer things.

Note that currently, the values of the arguments being passed to `innerFunction` are part of the **definition** of `outer`. In order to change those values we have to modify the `outer` function. This is not ideal.

With a simple change, something miraculous can happen. Rather than having `outer` return the result of calling `innerFunction`, let's have it return the function itself:

```
function outer(greeting, msg = "It's a fine day to learn") {
  const innerFunction = function (name, lang = "Python") {
    return `${greeting}, ${name}! ${msg} ${lang}`;
  };
  return innerFunction;
}
```

The return value of `outer` is now an **anonymous function**. To invoke it, we update the function call as follows:

```
outer("Hello")("student", "JavaScript");
//=> "Hello, student! It's a fine day to learn JavaScript"
```

The function call is processed by the JavaScript engine from left to right. First, `outer` is called with the argument "Hello." The return value of calling `outer("Hello")` is itself a function and, therefore, can itself be called. We do this by chaining on the second set of parentheses. This is basically the same concept as assigning a function expression to a variable and using the variable name followed by `()` to invoke the function. You can almost think of `outer("Hello")` as the "name" of the function that's returned by `outer`. It's the same as if we did this:

```
const storedFunction = outer("Hello");
storedFunction("student", "JavaScript");
//=> "Hello, student! It's a fine day to learn JavaScript"
```

Note that we are no longer calling `innerFunction` from inside `outer`. Amazingly, the code works ***exactly the same***: it ***still*** has access to those parent function's variables. It's like a little wormhole in space-time to the `outer`'s scope!

We can tighten this code up a bit more: instead of assigning the function expression to `innerFunction` and returning that, let's just return the function expression.

```
function outer(greeting, msg = "It's a fine day to learn") {
  return function (name, lang = "Python") {
    return `${greeting}, ${name}! ${msg} ${lang}`;
  };
}

outer("Hello")("student", "JavaScript");
//=> "Hello, student! It's a fine day to learn JavaScript"
```

To review: we first called `outer`, passing in the argument "Hello". `outer` **returned an anonymous function** inside which the default value of `msg` and the passed-in value of `greeting` have now been set. It's almost as if `outer` returned:

```
function(name, lang="Python") { // The "inner" function
  return `Hello, ${name}! It's a fine day to learn ${lang}`
}
```

We invoked this returned *"inner" function"* by adding the second set of parentheses and passing the arguments `"student"` and `"JavaScript"`, which were stored in `name` and `lang`. This filled in the final two values inside of the template string and returned:

```
"Hello, student! It's a fine day to learn JavaScript";
```

Define `Closure`

In the previous example, we could call the "inner" function, the **anonymous function**, a "closure." It "encloses" the function-level scope of its parent. And, like a backpack, it can carry out the knowledge that it saw — *even when you're out of the parent's scope*.

Recall the IIFE discussion. Since what's inside an IIFE can't be seen, if we wanted to let just tiny bits of information leak back out, we might want to pass that information back out, through a closure.

```
const array = (function (thingToAdd) {
  const base = 3;
  return [
    function () {
      return base + thingToAdd;
    },
    function () {
      return base;
    },
  ];
})(2);
```

Note that the value on the right of the `=` in the first line is a function expression. That function takes a single argument and returns an array that contains two functions. The `(2)` after the function expression executes that function (immediately), and the two inner functions are stored in the `array` variable.

Go ahead and copy the code above into your browser console and take a look at the values of the two elements of `array`. You should see the following:

```
array[0]; //=> ƒ () { return base + thingToAdd; }
array[1]; //=> ƒ () { return base; }
```

However, if you try looking at the value of `base` in the console you'll get a reference error: the value of `base` is not accessible outside the function it's defined in. Now go ahead and *call* the two returned functions; you should see the following:

```
array[0](); //=> 5
array[1](); //=> 3
```

The two functions being returned in `array` are **closures**; they have access to the `base` variable because it's defined in their parent function. When they're executed, they "let out" the values of the sum and the original base number, allowing us to see them.

Define `Scope Chain`

The mechanism behind all the cool stuff we just saw is the *scope chain* which allows functions defined inside functions (inside functions) to access all their parent (and grandparent) scopes' variables. Here's a simple example:

```
function demoChain(name) {
  const part1 = "hi";
  return function () {
    const part2 = "there";
    return function () {
      console.log(`${part1.toUpperCase()} ${part2} ${name}`);
    };
  };
}

demoChain("Dr. Stephen Strange")()(); //=> HI there Dr. Stephen Strange
```

When it is called, the innermost function has access to `name`, `part1`, and `part2` through the *scope chain*. As a result, when the `console.log()` statement is run, the string includes all three values. That's awesome wormhole, space-time magic!

LAB:

Implement a function called `saturdayFun`:

- It should define a function.
- It uses a default argument, `'roller-skate'` when no arguments are passed.
  - It allows the default argument to be overridden.

Implement a function called `mondayWork`:

- It should define a function.
- It uses a default argument, `'go to the office'` when no arguments are passed.
  - It allows the default argument to be overridden.

Implement a function called `wrapAdjective` :

- It should return a function. This "inner" function should:
  - take a single parameter that should default to `"special"` . Name it however you wish.
  - return a `String` of the form "You are ...!" where `...` should be the adjective this function received wrapped in visual flair.
- It should take as parameter a `String` that will be used to create visual flair.
- You may call the parameter whatever you like, but its default value should be `"*"` .
- Call example: `const encouragingPromptFunction = wrapAdjective("!!!")`

Thus a total call should be:

```
wrapAdjective("%")("a dedicated programmer"); //=> "You are %a dedicated programmer%!"
```

Run `npm test` to verify you've gotten this set of tests passing. Once you're done, commit and push your changes up to GitHub, then submit your work to Canvas using CodeGrade.

Conclusion

In this lesson, we've covered the basics of function declaration, invocation, and function scope. As a refresher on your skills, we've provided a simple lab to make sure that you're set for the new information coming up in the rest of this module.

Resources

- **Wikipedia — First-class function (https://en.wikipedia.org/wiki/First-class_function)**
- **StackOverflow — What is meant by 'first class object'? (https://stackoverflow.com/questions/705173/what-is-meant-by-first-class-object)**
- **Helephant — Functions are first class objects in javascript (Wayback Machine) (https://web.archive.org/web/20170606141950/http://helephant.com/2008/08/19/functions-are-first-class-objects-in-javascript/)**
- **2ality — Expressions versus statements in JavaScript (http://2ality.com/2012/09/expressions-vs-statements.html)**
- **MDN — Functions (https://developer.mozilla.org/en-US/docs/web/JavaScript/Reference/Operators/function)**
- **MDN — Statements and declarations (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements)**