**Function at Runtime**

**Introduction**

In this lesson we will learn what an object is, a function and a function object.

We will also learn how to write cleaner, more efficient code, and understand how powerful JavaScript can be in the application we build.

We will see how to:

* Pass function as arguments to other functions
* Identify when and how scope is created
* Keep track of the variables around using functions, a concept called closure
* Use functions that are immediately invoked to manage private data

**Functions are First-Class Functions**

In JavaScript, functions are first-class functions. This means that we can do with a function just anything that we can do with other data structures, such as numbers, strings, objects, arrays, etc.

JavaScript functions can:

* Be stored in variable
* Be returned from a function
* Be passed as an argument into another function

**Functions Can Return Functions (higher-order function)**

A function will always return one value (whether the value is explicitly specified in a return statement, or logging something to the console - undefined).

Since we know that functions are first-class functions, we can treat a function as a value and just return it from another function. A function that returns another function is known as a higher-order function.

function alertThenReturn() {

console.log('Message 1!');

return function () {

console.log('Message 2!');

};

}

alertThenReturn()

//Message 1!

var innerFunction = alertThenReturn();

innerFunction();

//Message 2!

alertTheReturn();

//Message 2!

When invoking the outer function, it returns “Message 1!”. It happens, because none of the code from the inner function is executed.

To have the inner function executed, we can either assign the function to a new variable - innerFunction - or invoke the function and add another set of parentheses - alertTheReturn().

**Callback Functions**

Because functions are first-class functions, we can also pass other functions as arguments into another function.

A function passed as an argument into another function is called a **callback function**.

Callback functions allow us to build our applications with composition, leading to cleaner and more efficient code.

function callAndAdd (n, callbackFunction) {

return n + callbackFunction;

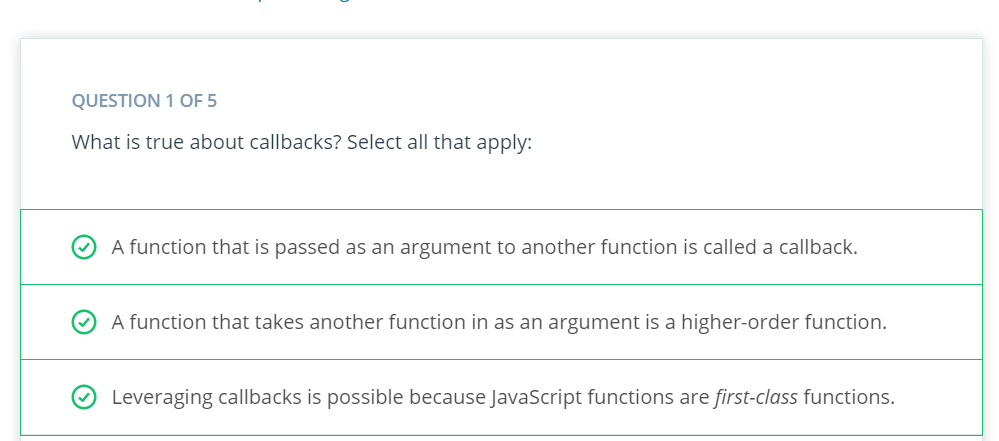
}

function returnsThree () {

return 3;

}

console.log(callAndAdd(50, returnsThree()));



function each(array, callback) {

for (let i = 0; i < array.length; i++) {

if (callback(array[i])) {

console.log(array[i]);

}

}

};

function isPositive(n) {

return n > 0;

};

r = each([-2, 7, 11, -4, -10], isPositive);

console.log(r);

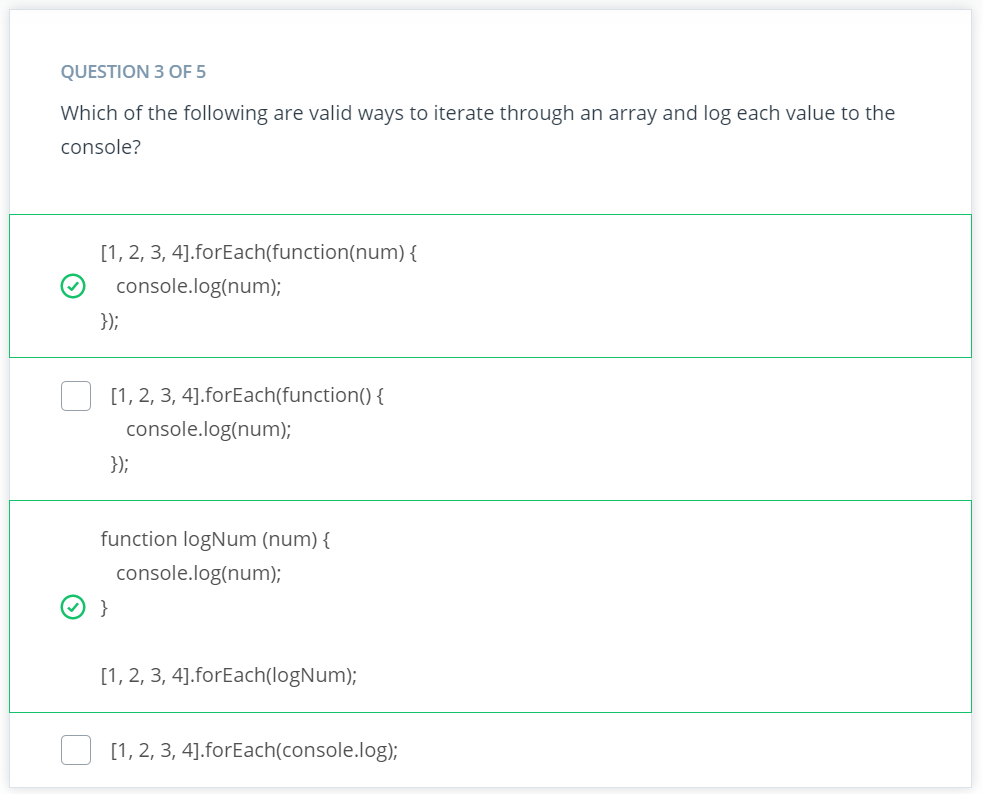
**Array Methods**

Array methods are an example of callback functions:

* forEach(): in the example below the function logIfOdd is passed as an argument to forEach(). Each element of the array is tested, and odd numbers are printed.
* [1, 5, 2, 4, 6, 3].forEach(function logIfOdd(n) {
* if (n % 2 !== 0) {
* console.log(n);
* }
* });

In case the function LogIfOdd is defined somewhere else, we could simply use the name of function into the forEach:

* [1, 5, 2, 4, 6, 3].forEach(logIfOdd});



* map(): this method operates similar to forEach() invoking a callback function for each element of the array. However, map() returns a new array, and does not modify the original array.

const names = ['David', 'Richard', 'Veronika'];

const nameLengths = names.map(function (callback) {

return callback.length

});

console.log(nameLengths);

//[5, 7, 7]

* filter(): this method is similar to the map() method: it is called on an array, it takes a function as an argument, and it returns a new array. The difference between them is that the filter() method uses the function passed as a test.

const names = ['David', 'Richard', 'Veronika'];

const shortNames = names.filter(function(name) {

return name.length < 6;

});

console.log(shortNames);

// ['David']

**Scope** - <https://dev.to/sandy8111112004/javascript-introduction-to-scope-function-scope-block-scope-d11>

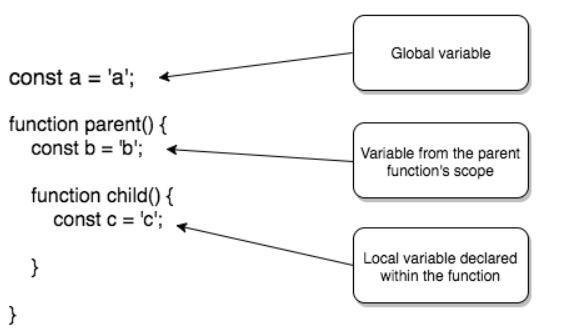
There are different types of scopes, which determines where a variable can be seen:

* Global scope: variables declared outside all the functions. Can be accessed in any other scopes.
* Block scope: variables within if, switch conditions, or for and while loops. {curly brackets} - const and let are used.
* Function scope: variables visible only within the function - var can be used.
* Lexical scope: when children have access to the variables defined in the parent scope, such as children functions.
* Runtime scope

**Function Access**

Functions have access to:

1. The function’s arguments
2. Local variables declared within the function
3. Variables from its parent function scope
4. Global variables (defined outside all functions)



const myName = 'Andrew';

// Global variable

function introduceMyself() {

const you = 'student';

// Variable declared where introduce() is defined

// (i.e., within introduce()'s parent function, introduceMyself())

function introduce() {

console.log(`Hello, ${you}, I'm ${myName}!`);

}

return introduce();

}

introduceMyself()

//Hello, student, I'm Andrew!

**JavaScript is Function-Scoped**

Variables in JavaScript are traditionally defined in the scope of a function, rather than in the scope of a block. Since entering a function will change scope, any variables defined inside that function are not available outside that function. On the other hand, if there are any variables defined inside a block (e.g., within an if statement), those variables are available outside that block.

/\*\*\*/

var globalNumber = 5;

function globalIncrementer() {

const localNumber = 10;

globalNumber += 1;

return globalNumber;

}

console.log(globalIncrementer());

// 6

console.log(globalIncrementer());

// 7

console.log(globalIncrementer());

// 8

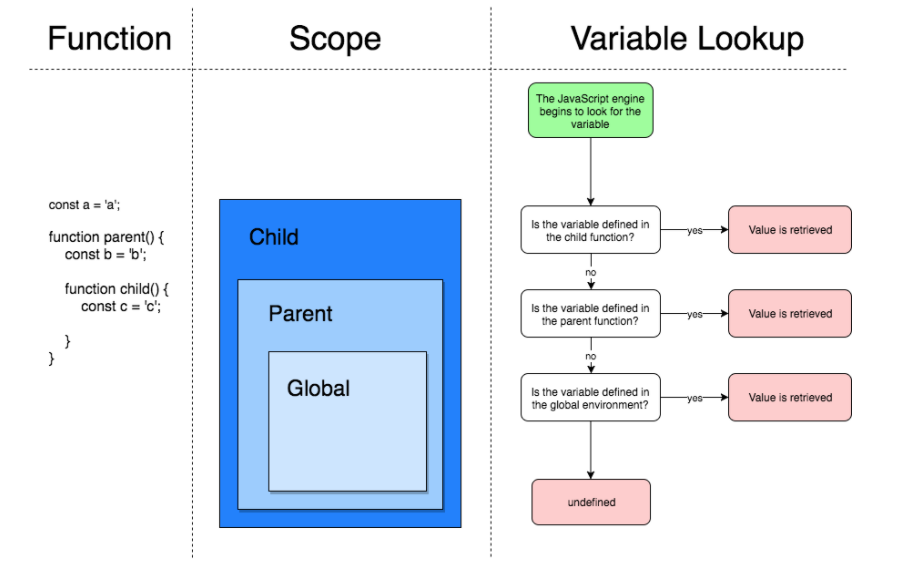
console.log(localNumber);

// ReferenceError: localNumber is not defined

Global variables can be accessed outside and inside a function, however a function scope variable can’t be accessed outside.

**Scope Chain**

Whenever our code attempts to access a variable during a function call, JavaScript interpreter will always start off by looking within its own local variables. If the variable isn’t found, the search will continue looking up what is called the scope chain.



When resolving a variable, the JavaScript engine begins by looking at the nested child function’s locally defined variables. If found, then the value is retrieved; if not, the JavaScript continues to look outward until the variable is resolved. If the JavaScript engine reaches the global scope and it is still unable to resolve the variable, the variable is undefined.

function one() {

two();

function two() {

three();

function three() {

// function three's code here

}

}

}

one();

In the above example, when one() function is called, all the other nested functions will be called as well (all the way to three()).

**Variable Shadowing**

When we create a variable with the same name as another variable somewhere in the scope chain, JavaScript shadows the variable in the outer scope.

Local variables that have the same name take precedence over those with a wider scope:

const symbol = '¥';//global variable

function displayPrice(price) {

const symbol = '$';//local variable

console.log(symbol + price);

}

displayPrice('80');

// '$80'

console.log(symbol)

// '¥'

**Functions Retain Their Scope**

Identifier lookup and the scope chain are really powerful tools for a function to access identifiers in the code, allowing us to create a function now, package it up with some variables, and save it to run later.

function remember(number) {

return function() {

return number;

}

}

const returnedFunction = remember(5);

console.log( returnedFunction() );

**Closure**

A closure is a combination of a function and it’s lexical environment (scope of the function - identifiers that can be accessed).

Every function has its own closure just like every function has its own scope.

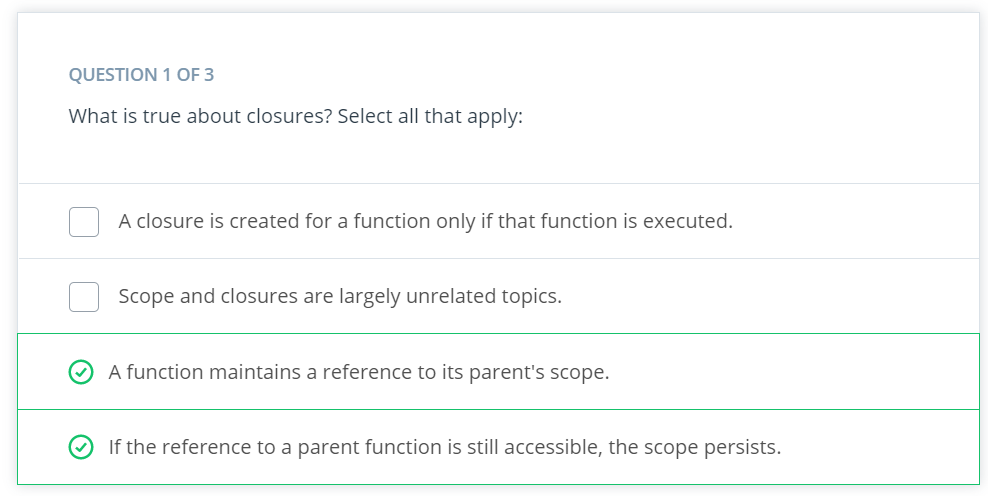
What makes a function’s closure powerful is when that function is returned from another function.

For example, when a function is declared within a function A, but returned in a function B.

**Creating a Closure**

Every time a function is defined, closure is created for that function.

Every function has a closure!



**Garbage Collection**

JavaScript manages memory with automatic garbage collection. When data is no longer referred to - there are no remaining references to that data available for executable code - it is garbage collected and will be destroyed at some later point in time. This frees up resources, such as memory.

**Function Declaration vs Function Expression**

* Function declaration: defines a function and does not require a variable to be assigned to it. It simply declares a function, and doesn’t itself return a value - needs to be invoked:

function returnHello() {

return 'Hello!';

}

* Function expression: returns a value. Function expressions can be anonymous or named, and are part of another expression’s syntax. Commonly assigned to variables

// anonymous

const myFunction = function () {

return 'Hello!';

};

// named

const myFunction = function returnHello() {

return 'Hello!';

};

**Immediately-Invoked Function Expression (IIFE): Structure and Syntax**

Immediately-invoked function expression is a function that is called immediately after it is defined.

//IIFE

(function sayHi(){

alert('Hi there!');

}

)();

// alerts 'Hi there!'

**Passing Arguments into IIFE’s**

The second pair of parentheses after the expression, not only immediately executes the function preceding it, but it is also the place to put any arguments that the function may need.

**IIFE’s and Private Scope**

One of the primary uses for IIFE is to create private scopes.

const myFunction = (

function () {

const hi = 'Hi!';

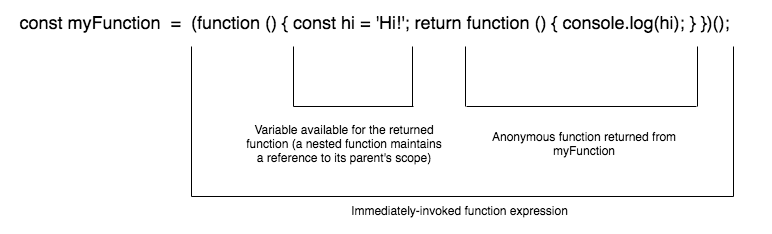
return function () {

console.log(hi);

}

}

)();



**IFFE’s, Private Scope, and Event Handling**

Using IIFE’s we can leverage a closure to protect a variable from being accessed externally. This prevents any accidental mutations or unwanted side-effects from inadvertently altering a particular variable.

**Benefits of Immediately-Invoked Function Expressions**

* Creates a private scope, protecting variables or methods from being accessed.
* Prevent the creation of extra global variables.
* Great for one-time tasks, it doesn’t pollute the global environment, hindering the chances of variable name collisions.