**Immediately Invoked Function Expression**

Welcome back to JavaScript Functions series. Earlier in this series we have covered different ways to define a function in JavaScript using Function Declaration, Function Expression and Anonymous Functions. In this article we will cover another aspect of JavaScript language – Immediately Invoked Function Expression [IIFE].

I hope by you now have a clear understanding that JavaScript language provides only two ways to define a function – using Function Declaration or Function Expression. Language features like anonymous functions, IIFE doesn’t provide new syntax to define a new function, rather they provides additional benefits to your program like callback functions, closures, public and private scope etc.

**IIFE - What’s in the name?**

Some people refer IIFE as *Self-executing anonymous functions* or *Self-invoked anonymous functions* and sometimes it gets totally confusing to follow these naming conventions. Just to be clear, all these names refer to same concept in JavaScript language. I prefer to call it as IIFE.

Before we dive into IIFE, we need to understand the concept of **Module** in JavaScript world.

**Build Module Using Functions**

A module is a collection of logically grouped functions to perform expected work in your program. The primary purpose of these modules is to hide the implementation details of functions and provides an entry point [or API] to provide an access these functions from outside. For example, you can define a *Payroll* module within your program which can contain functions like MakePayment, CalculateOverTime etc. It is not necessary to expose the internal implementation details of these functions as it might not be useful [or even dangerous!] to the external program.

In programming languages like C# or Java, a module can be defined using a class construct. Current version of JavaScript language doesn’t support class construct, however you can still define a module using JavaScript function as shown below.

var payroll = function() {

var MakePayment = function () {

console.log("In MakePayment function");

};

var CalculcateOverTime = function () {

console.log("In CalculateOverTime function");

};

};

Note that MakePayment, CalculateOverTime functions are declared inside a payroll function. Since JavaScript support only function scope, variables and functions declared inside a function [payroll in this case] are accessible only inside the function. In order to provide an entry point to payroll module, we need to change the function definition as shown below

var payroll = function() {

var MakePayment = function () {

console.log("In MakePayment function");

};

var CalculateOverTime = function () {

console.log("In CalculateOverTime function");

};

return {

MakePayment : MakePayment,

CalculateOverTime : CalculateOverTime

}

};

The return block in above function definition exposes both MakePayment and CalculateOverTime functions to external clients, which can be accessed as shown below

var payrollAdmin = payroll();

payrollAdmin.MakePayment();

payrollAdmin.CalculateOverTime();

With this approach you can even define a private variables or function within your module, which you can hide from external callers. In below code snippet, I have defined an audit function which is not exposed to the outside world.

var payroll = function() {

var audit = function(info) {

console.log("Auditing information " + info)

}

var makePayment = function () {

console.log("In MakePayment function");

audit("success");

};

var calculcateOverTime = function () {

console.log("In CalculateOverTime function");

audit("success");

};

return {

MakePayment : makePayment,

CalculateOverTime : calculcateOverTime

}

};

var payrollAdmin = payroll();

payrollAdmin.MakePayment();

payrollAdmin.CalculateOverTime();

With this basic introduction of modules in JavaScript, let’s understand ‘Global Variables’ and how to avoid them using IIFE.

**Global Variables**

In many programming languages using global variables is considered as a bad practice. In JavaScript they are evil and if you notice any global variables in your JavaScript program, you should try to refactor the code to remove them.

In the payroll module / function defined earlier, we have accidently created two global variables - payroll and payrollAdmin. In current example, since we know entire function definition we can ensure that global variables have no impact on function execution. However in complex and large scale JavaScript programs, if you create global variable like this, there are more chances that you override the definition of existing functions or module with your function. For e.g. if you refer external JavaScript file which already has payroll function in it and unfortunately it has been declared at global scope as well, your declared payroll function will override it during program execution, which may result into unexpected output. So, how can we eliminate global variables from our program? Enter IIFE

**IIFE**

An **IIFE** is a common JavaScript design pattern used by most popular libraries (jQuery, Backbone.js, Modernizr etc.) to place all library code inside of a local scope. It is just an anonymous function that is wrapped inside of a set of parentheses and invoked immediately.

A standard IIFE looks like this

(function () {

// Code goes here

})();

The advantage of the IIFE is that any vars declared inside it are inaccessible to the outside world. So how does that help us? The key is that an IIFE can have a return value just like any other function.

So let’s refactor our payroll function using IIFE syntax

var payroll = (function() {

var audit = function(info) {

console.log("Auditing information " + info)

}

var makePayment = function () {

console.log("In MakePayment function");

audit("success");

};

var calculcateOverTime = function () {

console.log("In CalculateOverTime function");

audit("success");

};

return {

MakePayment : makePayment,

CalculateOverTime : calculcateOverTime

}

}());

payroll.MakePayment();

payroll.CalculateOverTime();

As you can see now, we were able to use IIFE’s return value to provide access to MakePayment and CalculateOverTime functions without providing their internal implementation details.

I hope this article helped you to understand the JavaScript module and how you should encapsulate it using IIFE pattern to provide public / private scope for internal implementation.

Thanks for reading