**A Seminar Report**

**on**

**Asynchronous Programming in JavaScript**

In partial fulfillment of requirements for the degree of

Bachelor of Technology

In

Computer Science and Engineering

**SUBMITTED BY:**

**Saurabh Kumar**

**14MI539**

**COMPUTER SCIENCE AND ENGINEERING DEPARTMENT**

**NATIONAL INSTITUTE OF TECHNOLOGY**

**HAMIRPUR – 177005**

**MARCH 2017**

**ACKNOWLEDGEMENT**

**INDEX**

Abstract ......................................................................................... 4

Introduction ................................................................................... 5

Thread ........................................................................................... 6

Synchronous execution ................................................................. 6

Asynchronous execution ............................................................... 7

JavaScript ...................................................................................... 7

JavaScript Syntax .......................................................................... 8

How asynchronous code is written .............................................. 11

Event and Callbacks ..................................................................... 11

Promises ....................................................................................... 14

Advantages and disadvantages ..................................................... 16

References .................................................................................... 16

**FIGURE INDEX**

Figure 1 – States of Promises .................................................... 15

**Asynchronous Programming**

**Abstract**

Let us consider we have to perform some CPU intensive work to do in our program. For a multi threaded language it is not of much concern but in a single threaded language like JavaScript this turns out to be a big problem because in such a case the complete app will just freeze. JavaScript uses the concept of Asynchronous programming to get over this problem. Asynchronous programming is a means of parallel programming in which a unit of work runs separately from the main application thread and notifies the calling thread of its completion. The main benefits of asynchronous programming are improved application performance and responsiveness. It allows us to create web apps that works pleasantly and does not hang. It also enhances the performance of the web app.

JavaScript, the most famous language of the world right now, is extremely dependent on the concept of Asynchronous programming because the main application thread in JavaScript is single threaded. There are various ways that JavaScript establishes Asynchronous programming - Events, in which we register functions corresponding to certain events that can occur; Continuation passing style which uses the concept of callbacks in which the next step is explicitly passed as parameter to previous one. With the ever evolving nature of JavaScript, newer concepts like – Promises and Generators have made it easy to write asynchronous codes. Generators are pausable functions that can produce more than one value on request and Promises are just a name for the value that is not yet available but will be available in the future. Combining Generators and Promises, we can very easily write great asynchronous codes.

AJAX, Asynchronous JavaScript And XML, which help web apps to send requests to server without reloading the page uses the concept of asynchronous programming. Node,js, one of the famous back-end frameworks that implements JavaScript in the backend uses asynchronous programming to deal with all input and output that takes place in the system. Not only JavaScript, but famous languages like C# have frameworks that allows users to write asynchronous codes. Asynchronous programming is really the way to the future of coding.

**Asynchronous Programming**

**Introduction**

To understand what Asynchronous programming is and to compare it with other models, I would like to start off with a story. This is the story of one of my friend, let’s call him X. So, X loved cooking and always dreamt of having a restaurant of his own. For years, he kept saving money and finally he had enough money to open up a restaurant. So, he pursued his dream and bought a shop at a great place, he bought tables, chairs and every other thing he would need in the restaurant. In a few days, the restaurant is ready and he is ready to turn his dream into reality.

Initially, he was the only person working in the restaurant and it had only one counter. He will stand at the counter to take orders. People will queue up in front of the counter. The person at the front of the queue will place the order and after taking the order X will run back to the kitchen to get the order ready. During the time, X is preparing the order the complete queue will come to a halt and everybody will just wait for that order to come. Once X gets the food ready, he will run back to the counter pass the order to the one who ordered it and then the queue will finally move on. The next person will get to the front and same thing will take place again. This model is analogous to ‘**Synchronous single-threaded model**’ of programming in which as we can see the waiting time is quite high.

Soon enough X understood that the model is not working quite well. The waiting time for the model is very high and only a single order can be served at one time. ‘Serving more food at once would be a good idea’, he thought and so he hired new people. These new people are no ordinary person, they are as talented as X. Their job, just like X is to stand at the counter, take an order, run back to the kitchen to get the food ready, and then give it back to the one who ordered it. So, for them to work X creates a few new counters. Initially only the first counter is opened. When the need comes, counter-2, counter-3, etc are opened (of course the counter is always there but it starts processing orders only when need comes ) and people in queue are asked to move to different counters as per need. This way a maximum of n orders, when he had n counters, can be served. This increased the number of orders that could be served and reduced the waiting time. This is somewhat similar to ‘**Synchronous multithreaded model**’.

While the second model was working quite good, his innovative mind came up with a third model. For this model, he asked the people he hired to get a few days off and hired a few chefs who knew how to cook everything the restaurant served. Now, he removed all the counters except one. So, the restaurant is again back to one counter phase. Now, X stands at counter and people queue up. The first person will place the order but rather than waiting in front of the queue for his order to come, he/she is aked to move away from the queue and wait somewhere else. The order is passed to the kitchen where chef starts preparing the order and at the counter the queue moves ahead. The next person gets ahead and places the order and then moves away from front of the queue and this goes on. Once, the order is ready, the person who ordered it is called and is given the food. This works quite well. The salary and thus expenses of X is less than that of multi-talented people in previous model and further only one counter need to be maintained. The queue waits for none and even the waiting time is less. This is analogous to ‘**Asynchronous Programming**’.

When there’s only a single counter, it is ‘**Asynchronous single threaded**’ and when there’s more than one counter, it is ‘**Asynchronous Multi threaded**’.

In the three models every counter along with the person who stands on the counter is somewhat similar to a **Thread**. The people in the queue are the statements in the program being run. The chefs are somewhat similar to threads but are designated to perform only a single kind of operation.

Let’s talk about the technical details of all the terms we have seen so far.

**Thread**

A thread is a basic unit of CPU utilisation which consists of a thread ID, a Program Counter, a register set, and a stack. That’s all the things CPU needs to execute a set of instructions and so a thread can execute some program ot a part of program. When we talk about programming, threads basically are a way to execute some part of the program concurrently. To execute a program, we may spawn more than one threads and each thread can then execute some independent part of the complete program.

**Synchronous Execution**

In Sunchronous execution, the code is executed such that a statement will be executed only when all the statements before it is executed. The program is executed one statement after another and one will have to wait for it’s turn. It can be single-threaded and multi-threaded. In single threaded we have only one thread which executes the complete program and so all of the code is executed in a sequential manner, one statement after the other. In multi-threaded we have more than one threads and we can choose to execute some parts of the program in different threads. In every thread, the execution takes place in a sequential manner such that all previous statements must be executed before any statement.

**Asynchronous Execution**

The main thing about asynchronous execution is that ‘the queue in front of the main thread does not wait for the completion of a statement that does not require CPU’. For example – if we have some I/O code or event handling, the queue won’t wait and that code will be moved away from the main thread and the next statement in the queue continues with it’s execution. When the task that was moved away (the I/O operation or the event handling) is done with it’s job, or some triggering activity takes place, the main thread is notified about it. The main thread, when it’s gets free, than performs the required operation.

With Asynchronous execution, there’s no guarantee that a code will be executed in the order in which it is written. It may so happen that the output of a code written later in the program is shown before the code written earlier in the program and so care must be taken. Asynchronous programming, intenally may depend on multithreaded architecture but it’s not necessary. It may or may not spawn new threads

Now, we will go into the details of how Asynchronous code is written but before that I will like to give you a bit of introduction to JavaScript since the code that follows is in JavaScript.

**JavaScript**

* JavaScript is a high-level language that is used in almost every website that you have ever used or will use in the few years in future.
* It was created by Brendan Eich in 1995 and is one of the most famous programming language of present day.
* JavaScript is used in almost every website to add interactivity and make the website dynamic.
* Even though it was meant to be a language to be used only in browsers, it is not limited to websites only. These days it is used almost everywhere – as a server-side scripting language, to create desktop and mobile applications, etc.
* JavaScript syntax is inspired from C, C++, Java, etc but it has a mix of functional nature in it which makes it very powerful and easy to use.
* Further, the Object Oriented nature of JavaScript is very different from languages like C++ since it uses prototypes and until recent years did not have support for classes.
* JavaScript is developing day in and day out as a language and active work is being done on it.

**JavaScript Syntax**

* **Variables**
  + Variables in JavaScript are declared using the keyword var followed by a valid variable name and an optional value. This value can be one of the follwinfg types -
    - Undefined
    - Null
    - Boolean
    - Number
    - String
    - Object (Which consists of arrays as well)
  + We can declare Objects directly without declaring some class using curly brackets {}, further arrays can be declared using square brackets [].
  + When a variable is not initialised with any value, it is initialised with ‘undefined’ which is a special value;
  + JavaScript is Dynamically typed that means that when we declare a variable, we don’t need to specify it’s data type. The data type is evaluated at run time by the interpreter. Also, a variable that’s assigned a particular type of value can be re-assigned by a value of different type at a later time.
  + In the example that follows, we declare 5 variables. One that stores an integer value, one that stores a float value, one that stores a string, one that stores an array of strings and one that stores an object with two properties – name and createdBy.

var x = 10;

var y = 8.75;

var z = “Hello World”;

var a = [“This”, “is”, “an”, “array”];

var b = {

name: “JavaScript”,

createdBy: “Brendan Eich”

};

* **Functions**
  + Functions are the most basic unit of execution in JavaScript.
  + While declaring functions, we need not specify the return type of the function or the data type of the arguments it needs.
  + A function in JavaScript is declared using the ‘function’ keyword followed by an optional valid name followed by brackets containing arguments list and followed by curly braces containing all the statments.
  + A function may or may not return a value. When it does return a value it can be of any type – a number, a string, Array, String or even a function.

function func1(arg1, arg2){

// Some statements!

// Some more statements!

// Okay! Done!

}

* **Functions as first-class objects**
  + One thing that differentiates JavaScript from other languages like – C, C++, etc is the fact that functions in JavaScript is treated as ‘first-class objects’.
  + What functions as first-class objects mean is that everything that we can do with objects can be done with functions as well like – assigning functions to variables, assigning functions are properties of objects, declaring functions within other functions, passing functions as argument and returning function from other functions.
  + This allows us to write code in a functional style and is very important in writing asynchronous codes.

**Function assigned to variable :**

var func1 = function(){

// Some statements here

}

**Function declared within other function :**

function outerFunction(){

function innerFunction(){

// Some statements here

}

}

**Function taking function as argument :**

function func1(x){

// x is a function and will be run in the function

x();

}

function func2(){

// Some statements here

// This function will be passed to some other function

}

func1(func2);

**Function returning function :**

function foo(){

return function(){

// Some statements here

}

}

* **Single threaded nature of JavaScript**
  + JavaScript is a single threaded language i.e., there is only one thread that the JavaScript engine can spawn.
  + But even though the JavaScript engine has only one thread, it is not limited to using only that one. The environment in which the engine is being used can provide other specialised threads that could do only a single thing. Like – A browser can provide JavaScript engine with XHR thread that could make asynchronous AJAX requests, etc.
  + But the language itself can not spawn any new thread.

**How Asynchronous code is written?**

JavaScript uses two constructs to allow us to write asynchronous codes. These two constructs are:

1. Events and Callbacks
2. Promises

Events and Callbacks have been a part of language for long time now and they are supported in almost every platform whereas Promises are a new concept and are supported in only newer platforms.

* **Events and Callbacks**

**Events :** Events refer to some interaction that the user performs with the UI or website like - clicking a button, submitting a form or to something that occurs automatically in the browser like – the page getting completely loaded, etc. Adding some reaction to events is the main way of adding interactivity to pages.

**Event handlers / Event listeners :** Event handlers or Event listeners refer to functions that are assigned to some event and are executed only when that event occurs. We add event handlers to all the events of choice, whenever that event occurs either by user’s interaction or by browser, the respective event handler will be executed.

**Event and Asynchronousity**

Once we have registered an event handler for an event, the browser needs to keep waiting for the event to occur such that when the event occurs it can take appropriate action. But we need the browser to keep running other code while it is waiting for an event. This is where Asynchronous programming comes into play. Once an event handler is registered, one specialised thread builtin in browser keeps looking for events. When it founds that event has occured, it puts it’s respective handler into the queue and when the browser is free it takes one handler queue and runs it. This is how Events and assynchronousity are closely related.

**Callback**

* Callbacks basically are functions that are passed as an argument to other functions that execute those functions at a later time, maybe when some event happens like – completion of data read, etc.
* It is so called because functions passed as argument are called back whenever needed.
* There is nothing special about a callback function. Passing the function as an argument to other function makes it a callback. It needs not have any special characteristics to be such.

Until last year, callback was the only way by which asynchronousity can be implemented in JavaScript. Even today, this is the most supported and mostly used method.

**Basic Example**

//We define two functions – func1 and func2. Function func2 is passed as an // argument to function func1. So, func1 is a callback function!

function func1(x) {

// x is passed as an argument to this function

// x is a callback function

x();

}

function func2() {

// Do some awesome things here.

// This is a callback function!

}

func1(func2);

**Asynchronous Example**

// We add three event listeners – one for click, one for form submit and one for scroll

// someElement and aForm stores reference to a page element and a form.

someElement.addEventListener(“click”, function() {

// Do something when someElement is clicked

}, false);

aForm.addEventListener(“submit”, function() {

// Do something when aForm is submitted

// Like checking the values

}, false);

function pageScrolled() {

// User has scrolled the page.

// Perform some animations or do anything!

}

document.body.onscroll = pageScrolled;

**Another Asynchronous example**

// This example shows how we write code in case of Node.js. The example is rather // abstract one. We use a function execDBCommand. This function does not exists in // Node.js.

execDBCommand(‘db\_command’, function(err, data) {

if(err) {

// There’s some error. Perform error handling!

}

else {

// The transaction was successful.

// Data is available in data variable. Use it!

}

});

* **Promises**
  + Promises are a newer addition to JavaScript and are supported only in newer versions of broswers and Node.js.
  + Promise, as the name suggests, is a promise for a value that does not exist right now but may exist in the near future in the form of a resolved value or some error.
  + Even though, we need to write a bit more code than normal but Promises tend to simplify asynchronous codes by many folds.
  + **Creating Promises**
    - We create a new promise we use the built-in Promise constructor that takes a function called an **executor** as the argument.
    - The executor function takes two arguments – **resolve** and **reject**.
    - Within the executor we perform whatever operations we have to perform. Once the result from the operation is available we manually call resolve if it was successful but if it fails we need to manually call reject.
    - Doing so returns a promise. To this promise we need to add whatever function we want to execute in case of it successfully gets the value or in case we face an error. This is done using then function available on the promise

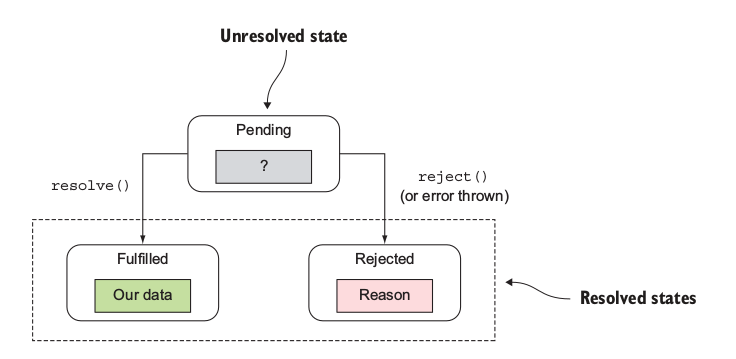
**Working of Promises**

When we call the promise constructor passing it the executor function, a promise is created. The executor function is immediately called but the promise produces value only when resolve or reject is called in the executor function. Even though, we can write a synchronous code in the executor function but this is not how it is mostly used. Mostly we will write some asynchronous code in the executor function like – making an AJAX request and when the result is available resolve the promise to pass it the response.

During it’s lifetime, a Promise passes through a couple of stages. When we create a promise, it starts off in **pending** state which simply means that the result of the asynchronous operation is still not available. Once, this result is available the promise will get into one of the two states – **Fulfilled** or **Rejected**. If the operation is successful and the data we need is produced, i.e. resolve function was called, promise goes to fulfilled state. On the other hand, if some error occurs during this process the promise, i.e. reject function is called, the promise goes to Rejected state.

Once, the promise has reached Fulfilled or Rejected state, there’s no turning back. It can not go back to the pending state or any other state.

Both the fulfilled and rejected states are collectively called **Resolved states** and represent the final state of the promise when it is done producing the value. Proper use of promise can simplify code very much and make it understandable and maintainable.

 **Figure 1. States of Promises**

**A Promise**

// Make a promise, a promise to return some value at a later time

var aPromise = new Promise(function(reject, resolve){

// Do something asynchronously

if(something\_is\_sucessfull) {

resolve(data\_produced);

}

else {

reject(error);

}

);

aPromise.then(function(data) {

// Promise is kept – some nice value has been returned

}, function(err) {

// Sadness pervails. We got an error.

// Do some error handling.

});

**Advantages and Disadvantages of Asynchronous Programming**

**Advantages:**

* Even a single thread could perform many operations.
* Better performance in terms of CPU utilisation.

**Disadvantages:**

Code complexity increases, that makes it pretty difficult to predict the flow of control.

**References**

* John Resig, Bear Bibeault, Josip Maras – Secrets of JavaScript Ninja, Manning Publications, 2nd Edition, 2016
* Trevor Burnham – Async JavaScript, The Pragmatic Programmers, 1st Edition, 2012