What is Callback?

In JavaScript, function is an object. Because of that, each function can hold another function as argument. As the result, we can write nested function. The function that have another function inside is called ***higher-order function*** this is similar to parent-child relationship. Therefore, higher-order function is parent. On the other hand, the child function, which is inside the parent function, called ***callback function***.

Callback Use Case

JavaScript is an ***event-driven language***, meaning that the list of function will be execute one after another. This characteristic of JavaScript is not a problem unless we want to work with another API. For example we have a function a() that needs to deal with a third party service. In running situation we have no idea how long will take for this third party API to be executed. If this API take two second to get ready, the JavaScript will not wait and go for other functions to run them. This will cause a problem. The callback can solve this problem.

The waiting time for API to receive a response can simulated by setTimeout predefined function. The setTimeout function create a delay for a function to execute.

```
function a(){
  // create waiting time for function A
  setTimeout( function(){
    console.log("This is function A");
  }, 2000 );
}

function b(){
  console.log("This is function B");
}

a();
b();
```

Figure 1 Function Order

Figure 1 illustrates two functions. First function will call with setTimeout delay of two seconds. As you can see we invoke function a() and then function b(), and we expect that console will run them in the order. Next figure showing the execution of these two functions in browser console.

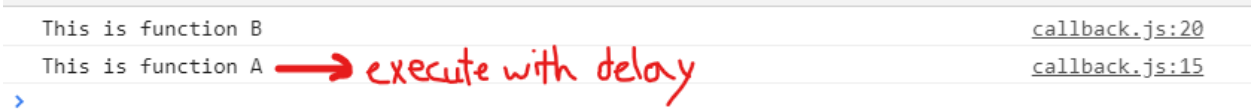

Figure 2 Console.log execution

As you can see the function b() appears first and with a two seconds delay function a() executed. The reason behind of this execution is the fact that JavaScript will not wait until the end of function a() to go through running function b(). If we want to execute function a() we need to use callback.

Callback is best way to make sure certain code does not execute until other code has already finished execution. In order to do that we need to create three changes to our figure1, which will illustrated in figure 3.

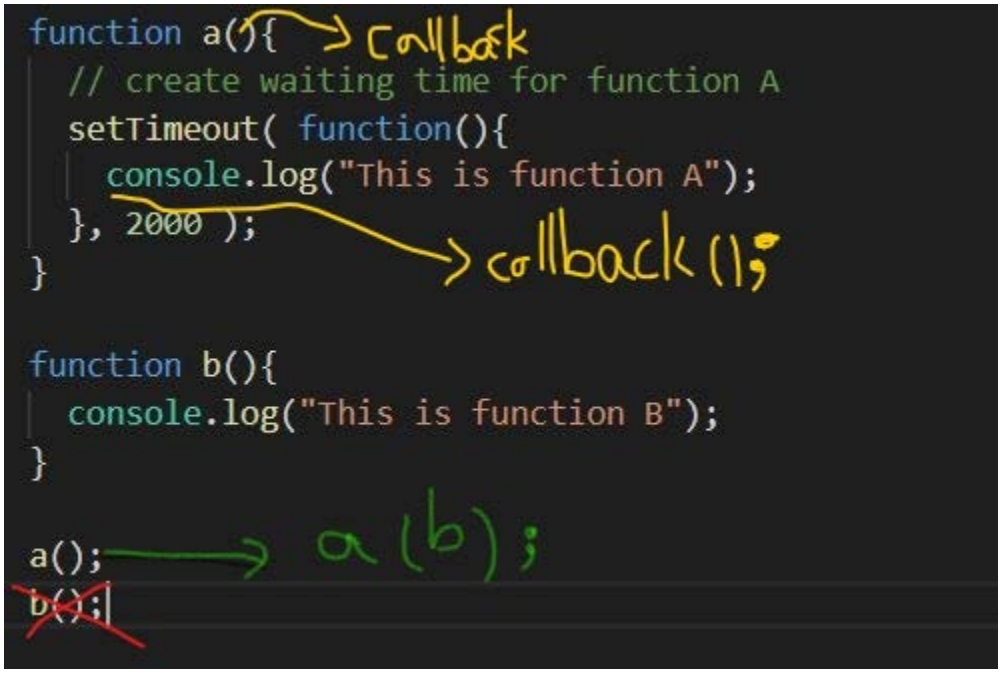

Figure 3 modify the code to execute a() before b()

As it shown there are three changes required to use callback inside the fiction a() and then we will call function b() from inside the first function by using callback() syntax.

The result should look like figure 4.

```
function a(callback){
 setTimeout( function(){
   console.log("This is function A");
   callback();
 }, 2000 );
}

function b(){
  console.log("This is function B");
}

a(b);
```

Figure 4 include callback to function a ()

As it appeared in figure 4, three changes has made in order to change the functionality of previous piece of code.

The next example shows how to add more argument in the higher-ordered function in figure 5 shows how we can use more than one argument beside the callback.

```
function StartEngine(engine, callback) {
  setTimeout( function(){
    console.log(`Starting the ${engine} now.`);
    callback();
  }, 2000);}

function setGear(){
  setTimeout( function(){
  console.log('go to first gear');
},1000 );}
StartEngine('BMW', setGear);
```

Figure 5How to have more than one argument in callback

It is important to mention that callback keyword should pass as latest argument in higher-ordered function. In this example as you can see the name of engine will pass to the first function with the name of second function that should be execute after first function.

Example of real-world scenario of Callback

1) First Case

“Asynchronous programming is essential when we develop any application because it avoids waiting in the main thread on long running operations such as disk I/O, network operations, database access, etc…”

In a normal case, if our program needs something to be done from the results of these long operations, our code is stuck until the operation is done and we proceed from that point.

Using the async mechanism, we can just trigger long running operations and continue with other tasks. These long running operations do the job in a different thread and when they complete it, they notify our main code, and our code can do the post actions from here. When we refer to our code, we are talking about our main thread which deals with the user interface or the thread that primarily processes a web request.

2) Second case

If you have a script that requests data from 3 different servers. Sometimes, depending on who knows what, the request to one of those servers may unexpectedly take too much time to execute. Imagine that it takes 10 seconds to get data from the second server. While you are waiting, the whole script is actually doing nothing. What if you could write a script that could, instead of waiting for the second request, simply skip it and start executing the third request, then go back to the second one, and proceed from where it left? That’s it. You minimize idle time by switching tasks.

Still, you don’t want to use an asynchronous code when you need a simple script, with little to no I/O.

One more important thing to mention is that all the code is running in a single thread. So if you expect that one part of the program will be executed in the background while your program will be doing something else, this won't happen.

3) Case of Blogpost

In the example we create a JavaScript object that create blog post, which illustrated in figure 6.

```
const posts = [
  {title: 'Post One', body: 'This is post one'},
  {title: 'Post Two', body: 'This is post two'}
];

function createPost(post, callback) {
  setTimeout(function() {
    posts.push(post);
    callback();
  }, 2000);
}


function getPosts() {
  setTimeout(function() {
    let output = '';
    posts.forEach(function(post){
      output += `<li>${post.title}</li>`;
    });
    document.body.innerHTML = output;
  }, 1000);
}

createPost({title: 'Post Three', body: 'This is post three'}, getPosts);
```

Figure 6 Blog Post

When we invoke createPost() we pass the callback function to it. So at the time that post has been created then the page is loaded.

Sdfsdf