JavaScript Callback Functions

Table of Contents

Callback functions are derived from a programming paradigm known as **functional programming**. At a fundamental level, functional programming specifies the use of functions as arguments. Functional programming was—and still is, though to a much lesser extent today—seen as an esoteric technique of specially trained, master programmers.

Fortunately, the techniques of functional programming have been elucidated so that mere mortals like you and me can understand and use them with ease. One of the chief techniques in functional programming happens to be *callback functions*. As you will read shortly, implementing callback functions is as easy as passing regular variables as arguments. This technique is so simple that I wonder why it is mostly covered in advanced JavaScript topics.

1. What is a Callback or Higher-order Function?

A callback function, also known as a higher-order function, is a function that is passed to another function (let's call this other function "otherFunction") as a parameter, and the callback function is called (or executed) inside the otherFunction. A callback function is essentially a pattern (an established solution to a common problem), and therefore, the use of a callback function is also known as a callback pattern.

Consider this common use of a callback function in jQuery:

```
//Note that the item in the click method's parameter is a function, not a
variable.

//The item is a callback function

$("#btn_1").click(function() {

alert("Btn 1 Clicked");

});
```

As you see in the preceding example, we pass a function as a parameter to the *click* method. And the click method will call (or execute) the callback function we passed to it. This example illustrates a typical use of callback functions in JavaScript, and one widely used in jQuery.

Ruminate on this other classic example of callback functions in basic JavaScript:

```
var friends = ["Mike", "Stacy", "Andy", "Rick"];



friends.forEach(function (eachName, index){

console.log(index + 1 + ". " + eachName); // 1. Mike, 2. Stacy, 3. Andy, 4. Rick

});
```

Again, note the way we pass an anonymous function (a function without a name) to the forEach method as a parameter.

So far we have passed anonymous functions as a parameter to other functions or methods. Let's now understand how callbacks work before we look at more concrete examples and start making our own callback functions.

2. How Callback Functions Work?

We can pass functions around like variables and return them in functions and use them in other functions. When we pass a callback function as an argument to another function, we are only passing the function definition. We are not executing the function in the parameter. In other words, we aren't passing the function with the trailing pair of executing parenthesis () like we do when we are executing a function.

And since the containing function has the callback function in its parameter as a function definition, it can execute the callback anytime.

Note that the callback function is not executed immediately. It is "called back" (hence the name) at some specified point inside the containing function's body. So, even though the first jQuery example looked like this:

```
//The anonymous function is not being executed there in the parameter.

//The item is a callback function

$("#btn_1").click(function() {

alert("Btn 1 Clicked");

});
```

the anonymous function will be called later inside the function body. Even without a name, it can still be accessed later via the *arguments* object by the containing function.

Callback Functions Are Closures

When we pass a callback function as an argument to another function, the callback is executed at some point inside the containing function's body just as if the callback were defined in the containing function. This means the callback is a closure. As we know, closures have access to the containing function's scope, so the callback function can access the containing functions' variables, and even the variables from the global scope.

3. Basic Principles when Implementing Callback Functions

While uncomplicated, callback functions have a few noteworthy principles we should be familiar with when implementing them.

Use Named OR Anonymous Functions as Callbacks

In the earlier jQuery and forEach examples, we used anonymous functions that were defined in the parameter of the containing function. That is one of the common patterns for using callback functions. Another popular pattern is to declare a named function and pass the name of that function to the parameter. Consider this:

```
// global variable
var allUserData = [];


// generic logStuff function that prints to console
function logStuff (userData) {
if ( typeof userData === "string")
{
 console.log(userData);
}
else if ( typeof userData === "object")
{
for (var item in userData) {
console.log(item + ": " + userData[item]);
}
}
```

```
}
// A function that takes two parameters, the last one a callback function
function getInput (options, callback) {
allUserData.push (options);
callback (options);
}
// When we call the getInput function, we pass logStuff as a parameter.
// So logStuff will be the function that will called back (or executed) inside the getInput function
getInput ({name:"Rich", speciality:"JavaScript"}, logStuff);
// name: Rich
// speciality: JavaScript
```

Pass Parameters to Callback Functions

Since the callback function is just a normal function when it is executed, we can pass parameters to it. We can pass any of the containing function's properties (or global properties) as parameters to the callback function. In the preceding example, we pass *options* as a parameter to the callback function. Let's pass a global variable and a local variable:

```
//Global variable
var generalLastName = "Clinton";

function getInput (options, callback) {
allUserData.push (options);
// Pass the global variable generalLastName to the callback function
callback (generalLastName, options);
}
```

Make Sure Callback is a Function Before Executing It

It is always wise to check that the callback function passed in the parameter is indeed a function before calling it. Also, it is good practice to make the callback function optional.

Let's refactor the getInput function from the previous example to ensure these checks are in place.

```
function getInput(options, callback) {

allUserData.push(options);



// Make sure the callback is a function

if (typeof callback === "function") {

// Call it, since we have confirmed it is callable

callback(options);

}

}
```

Without the check in place, if the getInput function is called either without the callback function as a parameter or in place of a function a non-function is passed, our code will result in a runtime error.

Problem When Using Methods With The this Object as Callbacks

When the callback function is a method that uses the *this* object, we have to modify how we execute the callback function to preserve the *this* object context. Or else the *this* object will either point to the global window object (in the browser), if callback was passed to a global function. Or it will point to the object of the containing method.

Let's explore this in code:

```
// Define an object with some properties and a method

// We will later pass the method as a callback function to another function

var clientData = {

id: 094545,

fullName: "Not Set",

// setUserName is a method on the clientData object

setUserName: function (firstName, lastName) {
```

```
// this refers to the fullName property in this object
this.fullName = firstName + " " + lastName;
}
}

function getUserInput(firstName, lastName, callback) {
// Do other stuff to validate firstName/lastName here

// Now save the names
callback (firstName, lastName);
}
```

In the following code example, when clientData.setUserName is executed, this.fullName will not set the fullName property on the clientData object. Instead, it will set fullName on the window object, since getUserInput is a global function. This happens because the *this* object in the global function points to the window object.

```
getUserInput ("Barack", "Obama", clientData.setUserName);

console.log (clientData.fullName);// Not Set

// The fullName property was initialized on the window object
console.log (window.fullName); // Barack Obama
```

Use the Call or Apply Function To Preserve this

We can fix the preceding problem by using the *Call* or *Apply* function (we will discuss these in a full blog post later). For now, know that every function in JavaScript has two methods: Call and Apply. And these methods are used to set the *this* object inside the function and to pass arguments to the functions.

Call takes the value to be used as the *this* object inside the function as the first parameter, and the remaining arguments to be passed to the function are passed individually (separated by commas of course). The **Apply** function's first parameter is also the value to be used as the *this* object inside the function, while the last parameter is an array of values (or the *arguments* object) to pass to the function.

This sounds complex, but lets see how easy it is to use Apply or Call. To fix the problem in the previous example, we will use the Apply function thus:

```
//Note that we have added an extra parameter for the callback object, called "callbackObj"
function getUserInput(firstName, lastName, callback, callbackObj) {
// Do other stuff to validate name here

// The use of the Apply function below will set the this object to be callbackObj
callback.apply (callbackObj, [firstName, lastName]);
}
```

With the *Apply* function setting the *this* object correctly, we can now correctly execute the callback and have it set the fullName property correctly on the clientData object:

```
// We pass the clientData.setUserName method and the clientData object as parameters. The clientData object will be used by the Apply function to set the this object
getUserInput ("Barack", "Obama", clientData.setUserName, clientData);

// the fullName property on the clientData was correctly set
console.log (clientData.fullName); // Barack Obama
```

We would have also used the *Call* function, but in this case we used the *Apply* function.

Multiple Callback Functions Allowed

We can pass more than one callback functions into the parameter of a function, just like we can pass more than one variable. Here is a classic example with jQuery's AJAX function:

```
function successCallback() {
// Do stuff before send
```

```
}

function successCallback() {
// Do stuff if success message received
}

function completeCallback() {
// Do stuff upon completion
}

function errorCallback() {
// Do stuff if error received
}

$.ajax({
url:"http://fiddle.jshell.net/favicon.png",
success:successCallback,
complete:completeCallback,
error:errorCallback

});
```

4. "Callback Hell" Problem And Solution

In asynchronous code execution, which is simply execution of code in any order, sometimes it is common to have numerous levels of callback functions to the extent that you have code that looks like the following. The messy code below is called callback hell because of the difficulty of following the code due to the many callbacks. I took this example from the node-mongodb-native, a MongoDB driver for Node.js. The example code below is just for demonstration:

```
var p_client = new Db('integration_tests_20', new Server("127.0.0.1", 27017,
```

```
{}), {'pk':CustomPKFactory});

p_client.open(function(err, p_client) {

p_client.dropDatabase(function(err, done) {

p_client.createCollection('test_custom_key', function(err, collection) {

collection.insert({'a':1}, function(err, docs) {

collection.find({'_id':new ObjectID("aaaaaaaaaaaa")}, function(err, cursor) {

cursor.toArray(function(err, items) {

test.assertEquals(1, items.length);


// Let's close the db

p_client.close();

});

});

});

});

});

});
```

You are not likely to encounter this problem often in your code, but when you do—and you will from time to time—here are two solutions to this problem.

1. Name your functions and declare them and pass just the name of the function as the callback, instead of defining an anonymous function in the parameter of the main function.
2. Modularity: Separate your code into modules, so you can export a section of code that does a particular job. Then you can import that module into your larger application.

5. Make Your Own Callback Functions

Now that you completely (I think you do; if not it is a quick reread :)) understand everything about JavaScript callback functions and you have seen that using callback functions are rather simple yet powerful, you should look at your own code for opportunities to use callback functions, for they will allow you to:

- Do not repeat code (DRY—Do Not Repeat Yourself)
- Implement better abstraction where you can have more generic functions that are versatile (can handle all sorts of functionalities)

- Have better maintainability
- Have more readable code
- Have more specialized functions.

It is rather easy to make your own callback functions. In the following example, I could have created one function to do all the work: retrieve the user data, create a generic poem with the data, and greet the user. This would have been a messy function with much if/else statements and, even still, it would have been very limited and incapable of carrying out other functionalities the application might need with the user data.

Instead, I left the implementation for added functionality up to the callback functions, so that the main function that retrieves the user data can perform virtually any task with the user data by simply passing the user's full name and gender as parameters to the callback function and then executing the callback function.

In short, the getUserInput function is versatile: it can execute all sorts of callback functions with myriad of functionalities.

```
// First, setup the generic poem creator function; it will be the callback
function in the getUserInput function below.

function genericPoemMaker(name, gender) {

console.log(name + " is finer than fine wine.");

console.log("Altruistic and noble for the modern time.");

console.log("Always admirably adorned with the latest style.");

console.log("A " + gender + " of unfortunate tragedies who still manages a
perpetual smile");

}



//The callback, which is the last item in the parameter, will be our
genericPoemMaker function we defined above.

function getUserInput(firstName, lastName, gender, callback) {

var fullName = firstName + " " + lastName;



// Make sure the callback is a function

if (typeof callback === "function") {

// Execute the callback function and pass the parameters to it

callback(fullName, gender);
```

```
}
```

```
}
```

Call the getUserInput function and pass the genericPoemMaker function as a callback:

```
getUserInput("Michael", "Fassbender", "Man", genericPoemMaker);
```

```
// Output
```

```
/* Michael Fassbender is finer than fine wine.
```

```
Altruistic and noble for the modern time.
```

```
Always admirably adorned with the latest style.
```

```
A Man of unfortunate tragedies who still manages a perpetual smile.
```

```
*/
```

Because the getUserInput function is only handling the retrieving of data, we can pass any callback to it. For example, we can pass a greetUser function like this:

```
function greetUser(customerName, sex) {
```

```
var salutation = sex && sex === "Man" ? "Mr." : "Ms.";
```

```
console.log("Hello, " + salutation + " " + customerName);
```

```
}
```

```

```

```
// Pass the greetUser function as a callback to getUserInput
```

```
getUserInput("Bill", "Gates", "Man", greetUser);
```

```

```

```
// And this is the output
```

```
Hello, Mr. Bill Gates
```

We called the same getUserInput function as we did before, but this time it performed a completely different task.

As you see, callback functions afford much versatility. And even though the preceding example is relatively simple, imagine how much work you can save yourself and how well abstracted your code will be if you start using callback functions. Go for it. Do it in the mornings; do it in the evenings; do it when you are down; do it when you are k

Note the following ways we frequently use callback functions in JavaScript, especially in modern web application development, in libraries, and in frameworks:

- For **asynchronous execution** (such as reading files, and making HTTP requests)
- In **Event Listeners/Handlers**
- In **setTimeout** and **setInterval** methods
- For **Generalization:** code conciseness

6. Conclusion

JavaScript callback functions are wonderful and powerful to use and they provide great benefits to your web applications and code. You should use them when the need arises; look for ways to refactor your code for **Abstraction**, **Maintainability**, and **Readability** with callback functions.

Source:

http://javascriptissexy.com/understand-javascript-callback-functions-and-use-them/