Chapter 29: Callbacks

Section 29.1: Simple Callback Usage Examples

Callbacks offer a way to extend the functionality of a function (or method) ***without changing*** its code. This approach is often used in modules (libraries / plugins), the code of which is not supposed to be changed.

Suppose we have written the following function, calculating the sum of a given array of values:

```
function foo(array) {
    var sum = 0;
    for (var i = 0; i < array.length; i++) {
        sum += array[i];
    }
    return sum;
}
```

Now suppose that we want to do something with each value of the array, e.g. display it using `alert()`. We could make the appropriate changes in the code of `foo`, like this:

```
function foo(array) {
    var sum = 0;
    for (var i = 0; i < array.length; i++) {
        alert(array[i]);
        sum += array[i];
    }
    return sum;
}
```

But what if we decide to use `console.log` instead of `alert()`? Obviously changing the code of `foo`, whenever we decide to do something else with each value, is not a good idea. It is much better to have the option to change our mind without changing the code of `foo`. That's exactly the use case for callbacks. We only have to slightly change `foo`'s signature and body:

```
function foo(array, callback) {
    var sum = 0;
    for (var i = 0; i < array.length; i++) {
        callback(array[i]);
        sum += array[i];
    }
    return sum;
}
```

And now we are able to change the behaviour of `foo` just by changing its parameters:

```
var array = [];
foo(array, alert);
foo(array, function (x) {
    console.log(x);
});
```

Examples with Asynchronous Functions

In jQuery, the `$.getJSON()` method to fetch JSON data is asynchronous. Therefore, passing code in a callback makes sure that the code is called *after* the JSON is fetched.

`$.getJSON()` syntax:

```
$.getJSON( url, dataObject, successCallback );
```

Example of `$.getJSON()` code:

```
$.getJSON("foo.json", {}, function(data) {
    // data handling code
});
```

The following would *not* work, because the data-handling code would likely be called *before* the data is actually received, because the `$.getJSON` function takes an unspecified length of time and does not hold up the call stack as it waits for the JSON.

```
$.getJSON("foo.json", {});
// data handling code
```

Another example of an asynchronous function is jQuery's `animate()` function. Because it takes a specified time to run the animation, sometimes it is desirable to run some code directly following the animation.

`.animate()` syntax:

```
jQueryElement.animate( properties, duration, callback );
```

For example, to create a fading-out animation after which the element completely disappears, the following code can be run. Note the use of the callback.

```
elem.animate( { opacity: 0 }, 5000, function() {
    elem.hide();
} );
```

This allows the element to be hidden right after the function has finished execution. This differs from:

```
elem.animate( { opacity: 0 }, 5000 );
elem.hide();
```

because the latter does not wait for `animate()` (an asynchronous function) to complete, and therefore the element is hidden right away, producing an undesirable effect.

Section 29.2: Continuation (synchronous and asynchronous)

Callbacks can be used to provide code to be executed after a method has completed:

```
/**
 * @arg {Function} then continuation callback
 */
function doSomething(then) {
  console.log('Doing something');
  then();
}

// Do something, then execute callback to log 'done'
doSomething(function () {
  console.log('Done');
});
```

```
console.log('Doing something else');

// Outputs:
//   "Doing something"
//   "Done"
//   "Doing something else"
```

The `doSomething()` method above executes synchronously with the callback - execution blocks until `doSomething()` returns, ensuring that the callback is executed before the interpreter moves on.

Callbacks can also be used to execute code asynchronously:

```
doSomethingAsync(then) {
  setTimeout(then, 1000);
  console.log('Doing something asynchronously');
}

doSomethingAsync(function() {
  console.log('Done');
});

console.log('Doing something else');

// Outputs:
//   "Doing something asynchronously"
//   "Doing something else"
//   "Done"
```

The `then` callbacks are considered continuations of the `doSomething()` methods. Providing a callback as the last instruction in a function is called a tail-call, which is optimized by ES2015 interpreters.

Section 29.3: What is a callback?

This is a normal function call:

```
console.log("Hello World!");
```

When you call a normal function, it does its job and then returns control back to the caller.

However, sometimes a function needs to return control back to the caller in order to do its job:

```
[1,2,3].map(function double(x) {
    return 2 * x;
});
```

In the above example, the function `double` is a callback for the function `map` because:

1. The function `double` is given to the function `map` by the caller.
2. The function `map` needs to call the function `double` zero or more times in order to do its job.

Thus, the function `map` is essentially returning control back to the caller every time it calls the function `double`. Hence, the name “callback”.

Functions may accept more than one callback:

```
promise.then(function onFulfilled(value) {
    console.log("Fulfilled with value " + value);
```

```
}, function onRejected(reason) {
    console.log("Rejected with reason " + reason);
});
```

Here then function `then` accepts two callback functions, `onFulfilled` and `onRejected`. Furthermore, only one of these two callback functions is actually called.

What's more interesting is that the function `then` returns before either of the callbacks are called. Hence, a callback function may be called even after the original function has returned.

Section 29.4: Callbacks and `this`

Often when using a callback you want access to a specific context.

```
function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', function() {
    console.log(this.msg);  // <= will fail because "this" is undefined
  });
}

var s = new SomeClass("hello", someElement);
```

Solutions

- Use `bind`

  `bind` effectively generates a new function that sets **`this`** to whatever was passed to `bind` then calls the original function.

  ```
  function SomeClass(msg, elem) {
    this.msg = msg;
    elem.addEventListener('click', function() {
      console.log(this.msg);
    }.bind(this));  // <-  bind the function to `this`
  }
  ```

- Use arrow functions

  Arrow functions automatically bind the current **`this`** context.

  ```
  function SomeClass(msg, elem) {
    this.msg = msg;
    elem.addEventListener('click',() => {   // <-  arrow function binds `this`
      console.log(this.msg);
    });
  }
  ```

Often you'd like to call a member function, ideally passing any arguments that were passed to the event on to the function.

Solutions:

- Use bind

  ```
  function SomeClass(msg, elem) {
  ```

```
  this.msg = msg;
  elem.addEventListener('click', this.handleClick.bind(this));
}

SomeClass.prototype.handleClick = function(event) {
  console.log(event.type, this.msg);
};
```

- Use arrow functions and the rest operator

```
function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', (...a) => this.handleClick(...a));
}

SomeClass.prototype.handleClick = function(event) {
  console.log(event.type, this.msg);
};
```

- For DOM event listeners in particular you can implement the `EventListener` interface

```
function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', this);
}

SomeClass.prototype.handleEvent = function(event) {
  var fn = this[event.type];
  if (fn) {
     fn.apply(this, arguments);
  }
};

SomeClass.prototype.click = function(event) {
  console.log(this.msg);
};
```

Section 29.5: Callback using Arrow function

Using arrow function as callback function can reduce lines of code.

The default syntax for arrow function is

```
() => {}
```

This can be used as callbacks

For example if we want to print all elements in an array [1,2,3,4,5]

without arrow function, the code will look like this

```
[1,2,3,4,5].forEach(function(x){
                console.log(x);
            }
```

With arrow function, it can be reduced to

```
[1,2,3,4,5].forEach(x => console.log(x));
```

Here the callback function `function(x){console.log(x)}` is reduced to `x=>console.log(x)`

Section 29.6: Error handling and control-flow branching

Callbacks are often used to provide error handling. This is a form of control flow branching, where some instructions are executed only when an error occurs:

```
const expected = true;

function compare(actual, success, failure) {
  if (actual === expected) {
    success();
  } else {
    failure();
  }
}

function onSuccess() {
  console.log('Value was expected');
}

function onFailure() {
  console.log('Value was unexpected/exceptional');
}

compare(true, onSuccess, onFailure);
compare(false, onSuccess, onFailure);

// Outputs:
//   "Value was expected"
//   "Value was unexpected/exceptional"
```

Code execution in `compare()` above has two possible branches: `success` when the expected and actual values are the same, and `error` when they are different. This is especially useful when control flow should branch after some asynchronous instruction:

```
function compareAsync(actual, success, failure) {
  setTimeout(function () {
    compare(actual, success, failure)
  }, 1000);
}

compareAsync(true, onSuccess, onFailure);
compareAsync(false, onSuccess, onFailure);
console.log('Doing something else');

// Outputs:
//   "Doing something else"
//   "Value was expected"
//   "Value was unexpected/exceptional"
```

It should be noted, multiple callbacks do not have to be mutually exclusive – both methods could be called. Similarly, the `compare()` could be written with callbacks that are optional (by using a noop as the default value - see Null Object pattern).