Callbacks: Using a Function as an Argument

Previously we explored how functions are first class objects, meaning they can be stored in variables just like any other value. In particular, we've been using built-in methods like `Array#forEach` and `Array#map` which accept (anonymous) functions as arguments. Now it's time to take a look under the hood and define our *own* functions that accept other functions as arguments.

When you finish reading this article, you should be able to define functions that accept callbacks.

What is a callback?

Defining a function that accepts another function as an argument is as simple as specifying a regular parameter. We'll name our parameter `callback` but you could very well name it whatever you please:

```
let foobar = function(callback) {
  console.log("foo");
  callback();
  console.log("bar");
};

let sayHello = function() {
  console.log("hello");
};

foobar(sayHello); // prints
// foo
// hello
// bar
```

A callback is always a function. In general, the callback is the function that is being *passed into* the other function. In the example above, `sayHello` is a callback, but `foobar` is not a callback. Notice that when we call `foobar(sayHello)`, we are not yet calling the `sayHello` function, instead we are passing the `sayHello` function itself into `foobar`. When execution enters the `foobar` function, the `callback` arg will refer to `sayHello`. This means that `callback()` will really evaluate to `sayHello()`.

n the example above we used a named callback, but we can also use a function expression directly. This is called an *anonymous* callback:

```
let foobar = function(callback) {
  console.log("foo");
  callback();
  console.log("bar");
};

foobar(function() {
  console.log("hello");
}); // prints
// foo
// hello
// bar
```

The advantage of using a named callback is that you can reuse the function many times, by referring to its name. Opt for an anonymous callback if you need a single-use.

A more interesting example

A callback behaves just like any other function, meaning it can accept it's own arguments and return a value. Let's define an `add` function that also accepts a callback:

```
let add = function(num1, num2, cb) {
  let sum = num1 + num2;
  let result = cb(sum);
  return result;
};

let double = function(num) {
  return num * 2;
};

let negate = function(num) {
  return num * -1;
};

console.log(add(2, 3, double)); // 10
console.log(add(4, 5, negate)); // -9
```

In the `add` function above, we pass the sum of `num1` and `num2` into the callback (`cb`) and return the result of the callback. Depending on the callback function we pass in, we can accomplish a wide range of behavior! This will come in handy when reusing code. A callback is just like a helper function, except now we can dynamically pass in *any* helper function.

To wrap things up, let's pass in some built-in functions and use them as callbacks. `Math.sqrt` is a function that takes in a number and returns its square root:

```
console.log(Math.sqrt(9)); // 3
console.log(Math.sqrt(25)); // 5
console.log(Math.sqrt(64)); // 8

let add = function(num1, num2, cb) {
  let sum = num1 + num2;
  let result = cb(sum);
  return result;
};

console.log(add(60, 4, Math.sqrt)); // 8
```

Refactoring for an optional callback

We have been claiming that we can leverage callbacks to write more versatile functions. However, a skeptic may argue that our previous `add` function is *not* so versatile because it can't return the normal sum without a trivial callback:

```
let add = function(num1, num2, cb) {
  let sum = num1 + num2;
  let result = cb(sum);
  return result;
};

// we just want the normal sum of 2 and 3
add(2, 3, function(n) {
  return n;
});
// this correctly returns the normal sum of 5, but the code is pretty gross
```

Have no fear! We can remedy this to have the best of both worlds, we just need a quick detour. JavaScript is not strict when it comes to passing too few arguments to a function. Here is an isolated example of this behavior:

```
let greet = function(firstName, lastName) {
  console.log("Hey " + firstName + "! Your last name is " + lastName + ".");
};

greet("Ada", "Lovelace"); // prints 'Hey Ada! Your last name is Lovelace.'
greet("Grace"); // prints 'Hey Grace! Your last name is undefined.'
```

If we pass too few arguments when calling a function, the parameters that do not have arguments will contain the value `undefined`. With that in mind, let's refactor our `add`function to *optionally*accept a callback:

```
let add = function(num1, num2, cb) {
  if (cb === undefined) {
    return num1 + num2;
  } else {
    return cb(num1 + num2);
  }
};

console.log(add(9, 40)); // 49
console.log(add(9, 40, Math.sqrt)); // 7
```

Amazing! As its name implies, our `add`function will return the plain old sum of the two numbers it is given. However, if it also passed a callback function, then it will utilize the callback too. A function that optionally accepts a callback is a fairly common pattern in JavaScript, so we'll be seeing this crop up on occasion.

What you've learned

- a callback is a function that is passed as an arg into another function
- we can pass named functions, anonymous functions, and even built-in functions as callbacks