Callback Functions

(https://github.com/learn-co-curriculum/phase-1-callback-functions) (https://github.com/learn-co-curriculum/phase-1-callback-functions/issues/new)

Learning Goals

- Understand that we can pass functions as arguments in JavaScript
- Define callback functions

Passing Functions as Arguments

We know we can pass numbers, strings, objects, and arrays into a function as arguments, but did you know we can also **pass functions into other functions**? We'll go into this in greater depth in an upcoming lesson, but it's important to start thinking about this concept now: in JavaScript, **functions are objects**. Specifically, they are objects with a special, hidden code property that can be invoked.

This is how we pass an object into a function:

```
function iReturnThings (thing) {
  return thing;
}

iReturnThings({ firstName: 'Brendan', lastName: 'Eich' });
// => {firstName: "Brendan", lastName: "Eich"}
```

And this is how we pass a function into a function:

```
iReturnThings(function () { return 4 + 5; });
// => f () { return 4 + 5; }
```

Notice that a representation of the passed-in function was returned, but **it was not invoked**. The `iReturnThings()` function accepted the passed-in function as its lone argument, `thing`. As with all arguments, `thing` was then available everywhere inside `iReturnThings()` as a local variable. When we passed a function into `iReturnThings()`, the `thing` variable contained that function. Currently, all `iReturnThings()` does is return whatever value is stored inside `thing`. However, if we know `thing` contains a function, we can do a piece of awesome, function-y magic to it: **we can invoke it** and return the function's result:

```
function iInvokeThings (thing) {
  return thing();
}

iInvokeThings(function () { return 4 + 5; });
// => 9

iInvokeThings(function () { return 'Hello, ' + 'world!'; });
// => "Hello, world!"
```

We pass in a function as the lone argument, store it inside the `thing` variable, and then use the invocation operator (a pair of parentheses) to invoke the stored function: `thing()`.

NOTE: As we dive deeper and deeper into functional programming in JavaScript, it bears repeating: this is **very** complicated material! This is likely the first time you're encountering any of this stuff so, if you're struggling with the new concepts, don't sweat it! Set aside some extra time to re-read and practice, and make sure you're coding along with every example we cover in the lessons.

Define Callback Functions

If you've done any outside reading on JavaScript, you've probably come across the name of the pattern we just introduced: *callback functions*. When we pass a function into another function wherein it might be invoked, we refer to the passed function as a *callback*. The term derives from the fact that the function isn't invoked immediately — instead it's *called back*, or invoked at a later point. (As an example, callback functions are commonly used to respond to user actions; we may define a callback to execute the appropriate code *when* the user clicks an element on the page.)

You may have noticed, but all of our callback functions so far have been *anonymous functions*; that is, we haven't assigned them an identifier. You're welcome to name your callback functions if you'd like, but generally, it just clutters things up if you only use the callback function in one place. And, anyway, we already have a way to refer to them: by the name of the parameter into which they're passed! For example:

```
function main (cb) {
  console.log(cb());
}

main(function () { return "After I get passed to the main() function as the only argument, I'm stored in the local 'cb' variable!"; });
// LOG: After I get passed to the main() function as the only argument, I'm stored in the local 'cb' variable!
```

1. We passed an anonymous function, `function () { return "After I get passed... }`, as the lone argument to our invocation of `main()`.
2. `main()` stored the passed-in function in the local `cb` variable and then invoked the callback function.
3. The invoked callback returned its long string, which was `console.log()`-ed out in `main()`.

We know that the parameters we define for our outer function are available anywhere inside the function. As a result, we can pass them as arguments to the callback function. For example:

```
function greet (name, cb) {
  return cb(name);
}

greet('Ada Lovelace', function (name) { return 'Hello there, ' + name; });
// => "Hello there, Ada Lovelace"

function doMath (num1, num2, cb) {
  return cb(num1, num2);
}
```

```
doMath(42, 8, function (num1, num2) { return num1 * num2; });
// => 336
```

In the above examples, what the `greet()` and `doMath()` functions are doing is pretty trivial: they're simply returning the result of calling the callback function. But let's consider another example. Imagine for a moment that we have a very expensive operation we need to execute, and that we need to do different things with the data it returns at different points in our program. We can use a callback to help us encapsulate that operation into its own function:

```
function somethingExpensive(cb) {
  // do something crazy,
  // like fetching a bajillion websites
  // then pass their data to the callback:
  cb(data);
}
```

This approach allows us to separate the execution of the expensive operation from the functions that use the data it returns. We do this by passing whichever function we currently need to `somethingExpensive()` as a callback. Once the expensive operation is finished, we simply call `cb()`, passing the data along as an argument.

Conclusion

In this lesson, we learned about JavaScript callback functions. If the topic feels a little abstract at this point, don't worry! We will learn a lot more about callback functions and how they can be used in upcoming lessons.

Resources

- **Programiz: JavaScript Callback Function (https://www.programiz.com/javascript/callback)**
- **StackOverflow: Explain Callbacks in Plain English (http://stackoverflow.com/questions/9596276/how-to-explain-callbacks-in-plain-english-how-are-they-different-from-calling-o)**