Chapter 20: Functional JavaScript

Section 20.1: Higher-Order Functions

In general, functions that operate on other functions, either by taking them as arguments or by returning them (or both), are called higher-order functions.

A higher-order function is a function that can take another function as an argument. You are using higher-order functions when passing callbacks.

```
function iAmCallbackFunction() {
    console.log("callback has been invoked");
}

function iAmJustFunction(callbackFn) {
    // do some stuff ...

    // invoke the callback function.
    callbackFn();
}

// invoke your higher-order function with a callback function.
iAmJustFunction(iAmCallbackFunction);
```

A higher-order function is also a function that returns another function as its result.

```
function iAmJustFunction() {
    // do some stuff ...

    // return a function.
    return function iAmReturnedFunction() {
        console.log("returned function has been invoked");
    }
}

// invoke your higher-order function and its returned function.
iAmJustFunction()();
```

Section 20.2: Identity Monad

This is an example of an implementation of the identity monad in JavaScript, and could serve as a starting point to create other monads.

Based on the conference by Douglas Crockford on monads and gonads

Using this approach reusing your functions will be easier because of the flexibility this monad provides, and composition nightmares:

```
f(g(h(i(j(k(value), j1), i2), h1, h2), g1, g2), f1, f2)
```

readable, nice and clean:

```
identityMonad(value)
    .bind(k)
    .bind(j, j1, j2)
    .bind(i, i2)
```

```
    .bind(h, h1, h2)
    .bind(g, g1, g2)
    .bind(f, f1, f2);
```

```
function identityMonad(value) {
    var monad = Object.create(null);

    // func should return a monad
    monad.bind = function (func, ...args) {
        return func(value, ...args);
    };

    // whatever func does, we get our monad back
    monad.call = function (func, ...args) {
        func(value, ...args);

        return identityMonad(value);
    };

    // func doesn't have to know anything about monads
    monad.apply = function (func, ...args) {
        return identityMonad(func(value, ...args));
    };

    // Get the value wrapped in this monad
    monad.value = function () {
        return value;
    };

    return monad;
};
```

It works with primitive values

```
var value = 'foo',
    f = x => x + ' changed',
    g = x => x + ' again';

identityMonad(value)
    .apply(f)
    .apply(g)
    .bind(alert); // Alerts 'foo changed again'
```

And also with objects

```
var value = { foo: 'foo' },
    f = x => identityMonad(Object.assign(x, { foo: 'bar' })),
    g = x => Object.assign(x, { bar: 'foo' }),
    h = x => console.log('foo: ' + x.foo + ', bar: ' + x.bar);

identityMonad(value)
    .bind(f)
    .apply(g)
    .bind(h); // Logs 'foo: bar, bar: foo'
```

Let's try everything:

```
var add = (x, ...args) => x + args.reduce((r, n) => r + n, 0),
    multiply = (x, ...args) => x * args.reduce((r, n) => r * n, 1),
```

```
    divideMonad = (x, ...args) => identityMonad(x / multiply(...args)),
    log = x => console.log(x),
    substract = (x, ...args) => x - add(...args);

identityMonad(100)
    .apply(add, 10, 29, 13)
    .apply(multiply, 2)
    .bind(divideMonad, 2)
    .apply(substract, 67, 34)
    .apply(multiply, 1239)
    .bind(divideMonad, 20, 54, 2)
    .apply(Math.round)
    .call(log); // Logs 29
```

Section 20.3: Pure Functions

A basic principle of functional programming is that it **avoids changing** the application state (statelessness) and variables outside its scope (immutability).

Pure functions are functions that:

- with a given input, always return the same output
- they do not rely on any variable outside their scope
- they do not modify the state of the application (**no side effects**)

Let's take a look at some examples:

> Pure functions must not change any variable outside their scope

Impure function

```
let obj = { a: 0 }

const impure = (input) => {
  // Modifies input.a
  input.a = input.a + 1;
  return input.a;
}

let b = impure(obj)
console.log(obj) // Logs { "a": 1 }
console.log(b) // Logs 1
```

The function changed the `obj.a` value that is outside its scope.

Pure function

```
let obj = { a: 0 }

const pure = (input) => {
  // Does not modify obj
  let output = input.a + 1;
  return output;
}
```

```
let b = pure(obj)
console.log(obj) // Logs { "a": 0 }
console.log(b) // Logs 1
```

The function did not change the object `obj` values

> Pure functions must not rely on variables outside their scope

Impure function

```
let a = 1;

let impure = (input) => {
  // Multiply with variable outside function scope
  let output = input * a;
  return output;
}

console.log(impure(2)) // Logs 2
a++; // a becomes equal to 2
console.log(impure(2)) // Logs 4
```

This **impure** function rely on variable `a` that is defined outside its scope. So, if a is modified, `impure`'s function result will be different.

Pure function

```
let pure = (input) => {
  let a = 1;
  // Multiply with variable inside function scope
  let output = input * a;
  return output;
}

console.log(pure(2)) // Logs 2
```

The `pure`'s function result **does not rely** on any variable outside its scope.

Section 20.4: Accepting Functions as Arguments

```
function transform(fn, arr) {
    let result = [];
    for (let el of arr) {
        result.push(fn(el)); // We push the result of the transformed item to result
    }
    return result;
}

console.log(transform(x => x * 2, [1,2,3,4])); // [2, 4, 6, 8]
```

As you can see, our `transform` function accepts two parameters, a function and a collection. It will then iterate the collection, and push values onto the result, calling `fn` on each of them.

Looks familiar? This is very similar to how `Array.prototype.map()` works!

```
console.log([1, 2, 3, 4].map(x => x * 2)); // [2, 4, 6, 8]
```