Chapter 02: First Class Functions

A Quick Review

When we say functions are "first class", we mean they are just like everyone else... so in other words a normal class. We can treat functions like any other data type and there is nothing particularly special about them - they may be stored in arrays, passed around as function parameters, assigned to variables, and what have you.

That is JavaScript 101, but worth mentioning since a quick code search on github will reveal the collective evasion, or perhaps widespread ignorance of this concept. Shall we go for a feigned example? We shall.

```
const hi = name => `Hi ${name}`;
const greeting = name => hi(name);
```

Here, the function wrapper around `hi` in `greeting` is completely redundant. Why? Because functions are *callable* in JavaScript. When `hi` has the `()` at the end it will run and return a value. When it does not, it simply returns the function stored in the variable. Just to be sure, have a look yourself:

```
hi; // name => `Hi ${name}`
hi("jonas"); // "Hi jonas"
```

Since `greeting` is merely in turn calling `hi` with the very same argument, we could simply write:

```
const greeting = hi;
greeting("times"); // "Hi times"
```

In other words, `hi` is already a function that expects one argument, why place another function around it that simply calls `hi` with the same bloody argument? It doesn't make any damn sense. It's like donning your heaviest parka in the dead of July just to blast the air and demand an ice lolly.

It is obnoxiously verbose and, as it happens, bad practice to surround a function with another function merely to delay evaluation (we'll see why in a moment, but it has to do with maintenance)

A solid understanding of this is critical before moving on, so let's examine a few more fun examples excavated from the library of npm packages.

```
// ignorant
const getServerStuff = callback => ajaxCall(json => callback(json));

// enlightened
const getServerStuff = ajaxCall;
```

The world is littered with ajax code exactly like this. Here is the reason both are equivalent:

```
// this line
ajaxCall(json => callback(json));

// is the same as this line
ajaxCall(callback);

// so refactor getServerStuff
const getServerStuff = callback => ajaxCall(callback);

// ...which is equivalent to this
const getServerStuff = ajaxCall; // <-- look mum, no ()'s
```

And that, folks, is how it is done. Once more so that we understand why I'm being so persistent.

```
const BlogController = {
  index(posts) { return Views.index(posts); },
  show(post) { return Views.show(post); },
  create(attrs) { return Db.create(attrs); },
  update(post, attrs) { return Db.update(post, attrs); },
  destroy(post) { return Db.destroy(post); },
};
```

This ridiculous controller is 99% fluff. We could either rewrite it as:

```
const BlogController = {
  index: Views.index,
  show: Views.show,
  create: Db.create,
  update: Db.update,
  destroy: Db.destroy,
};
```

… or scrap it altogether since it does nothing more than just bundle our Views and Db together.

Why Favor First Class?

Okay, let's get down to the reasons to favor first class functions. As we saw in the `getServerStuff` and `BlogController` examples, it's easy to add layers of indirection that provide no added value and only increase the amount of redundant code to maintain and search through.

In addition, if such a needlessly wrapped function must be changed, we must also need to change our wrapper function as well.

```
httpGet('/post/2', json => renderPost(json));
```

If `httpGet` were to change to send a possible `err`, we would need to go back and change the "glue".

```
// go back to every httpGet call in the application and explicitly pass err along.
httpGet('/post/2', (json, err) => renderPost(json, err));
```

Had we written it as a first class function, much less would need to change:

```
// renderPost is called from within httpGet with however many arguments it wants
httpGet('/post/2', renderPost);
```

Besides the removal of unnecessary functions, we must name and reference arguments. Names are a bit of an issue, you see. We have potential misnomers - especially as the codebase ages and requirements change.

Having multiple names for the same concept is a common source of confusion in projects. There is also the issue of generic code. For instance, these two functions do exactly the same thing, but one feels infinitely more general and reusable:

```js
// specific to our current blog
const validArticles = articles =>
  articles.filter(article => article !== null && article !== undefined),

// vastly more relevant for future projects
const compact = xs => xs.filter(x => x !== null && x !== undefined);
```

By using specific naming, we've seemingly tied ourselves to specific data (in this case `articles`). This happens quite a bit and is a source of much reinvention.

I must mention that, just like with Object-Oriented code, you must be aware of `this` coming to bite you in the jugular. If an underlying function uses `this` and we call it first class, we are subject to this leaky abstraction's wrath.

```js
const fs = require('fs');

// scary
fs.readFile('freaky_friday.txt', Db.save);

// less so
fs.readFile('freaky_friday.txt', Db.save.bind(Db));
```

Having been bound to itself, the `Db` is free to access its prototypical garbage code. I avoid using `this` like a dirty nappy. There's really no need when writing functional code. However, when interfacing with other libraries, you might have to acquiesce to the mad world around us.

Some will argue that `this` is necessary for optimizing speed. If you are the micro-optimization sort, please close this book. If you cannot get your money back, perhaps you can exchange it for something more fiddly.

And with that, we're ready to move on.

Chapter 03: Pure Happiness with Pure Functions