Chapter 01: What Ever Are We Doing?

Introductions

Hi there! I'm Professor Franklin Frisby. Pleased to make your acquaintance. We'll be spending some time together, as I'm supposed to teach you a bit about functional programming. But enough about me, what about you? I'm hoping that you're at least a bit familiar with the JavaScript language, have a teensy bit of Object-Oriented experience, and fancy yourself a working class programmer. You don't need to have a PhD in Entomology, you just need to know how to find and kill some bugs.

I won't assume that you have any previous functional programming knowledge, because we both know what happens when you assume. I will, however, expect you to have run into some of the unfavorable situations that arise when working with mutable state, unrestricted side effects, and unprincipled design. Now that we've been properly introduced, let's get on with it.

The purpose of this chapter is to give you a feel for what we're after when we write functional programs. In order to be able to understand the following chapters, we must have some idea about what makes a program *functional*. Otherwise we'll find ourselves scribbling aimlessly, avoiding objects at all costs - a clumsy endeavor indeed. We need a clear bullseye to hurl our code at, some celestial compass for when the waters get rough.

Now, there are some general programming principles - various acronymic credos that guide us through the dark tunnels of any application: DRY (don't repeat yourself), YAGNI (ya ain't gonna need it), loose coupling high cohesion, the principle of least surprise, single responsibility, and so on.

I won't belabor you by listing each and every guideline I've heard throughout the years… The point of the matter is that they hold up in a functional setting, although they're merely tangential to our ultimate goal. What I'd like you to get a feel for now, before we get any further, is our intention when we poke and prod at the keyboard; our functional Xanadu.

A Brief Encounter

Let's start with a touch of insanity. Here is a seagull application. When flocks conjoin they become a larger flock, and when they breed, they increase by the number of seagulls with whom they're breeding. Now, this is not intended to be good Object-Oriented code, mind you, it is here to highlight the perils of our modern, assignment based approach. Behold:

```
class Flock {
  constructor(n) {
    this.seagulls = n;
  }

  conjoin(other) {
    this.seagulls += other.seagulls;
    return this;
  }

  breed(other) {
    this.seagulls = this.seagulls * other.seagulls;
    return this;
  }
}

const flockA = new Flock(4);
const flockB = new Flock(2);
const flockC = new Flock(0);
const result = flockA
  .conjoin(flockC)
  .breed(flockB)
  .conjoin(flockA.breed(flockB))
  .seagulls;
// 32
```

Who on earth would craft such a ghastly abomination? It is unreasonably difficult to keep track of the mutating internal state. And, good heavens, the answer is even incorrect! It should have been `16`, but `flockA` wound up permanently altered in the process. Poor `flockA`. This is anarchy in the I.T.! This is wild animal arithmetic!

If you don't understand this program, it's okay, neither do I. The point to remember here is that state and mutable values are hard to follow, even in such a small example.

Let's try again, this time using a more functional approach:

```
const conjoin = (flockX, flockY) => flockX + flockY;
const breed = (flockX, flockY) => flockX * flockY;

const flockA = 4;
const flockB = 2;
const flockC = 0;
const result =
    conjoin(breed(flockB, conjoin(flockA, flockC)), breed(flockA, flockB));
// 16
```

Well, this time we got the right answer. With much less code. The function nesting is a tad confusing… (we'll remedy this situation in ch5). It's better, but let's dig a little bit deeper. There

are benefits to calling a spade a spade. Had we scrutinized our custom functions more closely, we would have discovered that we're just working with simple addition (`conjoin`) and multiplication (`breed`).

There's really nothing special at all about these two functions other than their names. Let's rename our custom functions to `multiply` and `add` in order to reveal their true identities.

```
const add = (x, y) => x + y;
const multiply = (x, y) => x * y;

const flockA = 4;
const flockB = 2;
const flockC = 0;
const result =
    add(multiply(flockB, add(flockA, flockC)), multiply(flockA, flockB));
// 16
```

And with that, we gain the knowledge of the ancients:

```
// associative
add(add(x, y), z) === add(x, add(y, z));

// commutative
add(x, y) === add(y, x);

// identity
add(x, 0) === x;

// distributive
multiply(x, add(y,z)) === add(multiply(x, y), multiply(x, z));
```

Ah yes, those old faithful mathematical properties should come in handy. Don't worry if you didn't know them right off the top of your head. For a lot of us, it's been a while since we learned about these laws of arithmetic. Let's see if we can use these properties to simplify our little seagull program.

```
// Original line
add(multiply(flockB, add(flockA, flockC)), multiply(flockA, flockB));

// Apply the identity property to remove the extra add
// (add(flockA, flockC) == flockA)
add(multiply(flockB, flockA), multiply(flockA, flockB));

// Apply distributive property to achieve our result
multiply(flockB, add(flockA, flockA));
```

Brilliant! We didn't have to write a lick of custom code other than our calling function. We include `add` and `multiply` definitions here for completeness, but there is really no need to write them - we surely have an `add` and `multiply` provided by some existing library.

You may be thinking "how very strawman of you to put such a mathy example up front". Or "real programs are not this simple and cannot be reasoned about in such a way." I've chosen this example because most of us already know about addition and multiplication, so it's easy to see how math is very useful for us here.

Don't despair - throughout this book, we'll sprinkle in some category theory, set theory, and lambda calculus and write real world examples that achieve the same elegant simplicity and results as our flock of seagulls example. You needn't be a mathematician either. It will feel natural and easy, just like you were using a "normal" framework or API.

It may come as a surprise to hear that we can write full, everyday applications along the lines of the functional analog above. Programs that have sound properties. Programs that are terse, yet easy to reason about. Programs that don't reinvent the wheel at every turn. Lawlessness is good if you're a criminal, but in this book, we'll want to acknowledge and obey the laws of math.

We'll want to use a theory where every piece tends to fit together so politely. We'll want to represent our specific problem in terms of generic, composable bits and then exploit their properties for our own selfish benefit. It will take a bit more discipline than the "anything goes" approach of imperative programming (we'll go over the precise definition of "imperative" later in the book, but for now consider it anything other than functional programming). The payoff of working within a principled, mathematical framework will truly astound you.

We've seen a flicker of our functional northern star, but there are a few concrete concepts to grasp before we can really begin our journey.

Chapter 02: First Class Functions