Thinking With Types

```
def wc(d: Document): Map[Word, Int] =
    countWordOccurrences(documentToWords(d))

splitDocumentIntoParagraphs(d: Document):
    Seq[Paragraph] = ???

splitParagraphIntoWords(p: Paragraph):
    Seq[Word] = ???
```

Alvin Alexander

Thinking With Types

Alvin J. Alexander

An introduction to
functional thinking in Scala

Copyright

Thinking With Types

Version 0.1, published July 26, 2020
Version 0.2, published October 10, 2021

Other books by Alvin Alexander:

- Scala Cookbook, 2nd Edition
- Functional Programming, Simplified
- How I Sold My Business: A Personal Diary
- A Survival Guide for New Consultants

Contents

1
Introduction

This booklet is created from a short series of blog posts that I titled, *Thinking With Types*. It could also be called, *Programming With Types*.

1.1 The TL;DR summary

Unless you're a library writer, when you program in Java you often don't work with very complicated types or combinations of types. For example, using the "programmer categories" that Martin Odersky defined in 2011:

Application Programmer	Library Designer
Beginning [**A1**]	
Intermediate [**A2**]	Junior [**L1**]
Expert [**A3**]	Senior [**L2**]
	Expert [**L3**]

I think it's accurate to say that Java *library designers* in the categories L2, L3, and possibly L1 work with complex data types, but *application programmers* in the categories A1, A2, and even A3 don't often get into complex types.

Conversely, if you get into functional programming in Scala — what I call Scala/FP — you'll be working with data types all the time, even in the A1-A3 categories. A key here — as you'll see — is that this isn't a bad thing, and indeed, it can be very, very good for your code.

> When I write *Java/OOP*, I mean Java used as an object-oriented programming (OOP) language. You're also more than welcome to use Scala/OOP — Scala as an OOP language — but this booklet focuses on writing pure functions and Scala/FP.

2

Java and object-oriented programming

In Java I rarely used generic data types, and rarely thought about types much at all. To demonstrate why, I'll go back to one of my favorite examples, writing code for a pizza store.

> Please note that my Java code may not be 100% correct. I haven't put this in an IDE or tried to compile it. I'm just trying to quickly demonstrate the OOP approach.

To start modeling a point-of-sales application for a pizza store, I'll first need my usual enumerations:

```
enum Topping {
    Cheese,
    Pepperoni,
    BlackOlives
}

enum CrustSize {
    Small,
    Medium,
    Large
}

enum CrustType {
    Regular,
    Thin,
    Thick
}
```

Given those, let's create a `Pizza` class. But before doing that, let's list the attributes and behaviors of an OOP pizza. The attributes are basically what I just listed with those enums:

- Crust size
- Crust type
- Types of toppings

The behaviors correspond to those attributes:

- Add topping
- Remove topping
- Set crust size
- Set crust type

If I also kept price-related information, a pizza could also calculate its own price, but since the toppings and crust attributes don't carry price information with them, that's pretty much all of the behaviors of a `Pizza`.

Having gone through this exercise several times before, I know that I want all of the food-related items in a pizza store — pizza, breadsticks, soda, etc. — to extend a base `Product` class:

```
public interface Product {}
```

In the real world a `Product` will have attributes like cost and potentially a sales price associated with it, but for the purposes of this exercise I'll just leave it as a marker interface like that.

Given that, and the attributes and behaviors of a pizza, my initial code for a Java/OOP `Pizza` class looks like this:

```
public class Pizza implements Product {

    private List<Topping> toppings = new ArrayList<>();
    private CrustSize crustSize;
    private CrustType crustType;
```

```
    public Pizza(CrustSize crustSize, CrustType crustType) {
        this.crustSize = crustSize;
        this.crustType = crustType;
    }

}
```

That code is followed by a series of getter and setter methods that have these type signatures:

```
public CrustSize getCrustSize()
public CrustType getCrustType()
public List<Topping> getToppings()

public void setCrustSize(CrustSize crustSize)
public void setCrustType(CrustType crustType)
public void setToppings(List<Topping> toppings)
```

The thing to notice here is that this is all very simple code. The getters don't take any input parameters and they return basic data types, and the setters all return `void`. I wrote Java code like this for more than 12 years.

In fact, if I wrote more code, such as for an `Order` class, you'd see that the pattern is the same, all very simple code:

```
public class Order {

    private List<Product> lineItems = new ArrayList<>();
    public Order() {}

    public void addItem(Product p)
    public void removeItem(Product p)
    public List<Product> getItems()

    public String getPrintableReceipt()
    public BigDecimal getTotalPrice()

}
```

The closest I get to using generic types in that code is this:

```
private List<Topping> toppings = new ArrayList<>();
private List<Product> lineItems = new ArrayList<>();
```

2.0.1 Summary, Part 1

To summarize what you saw in these examples:

- My Java/OOP "pizza" code did not use any generic types
- A lot of methods return `void`
- A lot of methods have no input parameters
- I *always* use mutable data structures

2.0.2 Summary, Part 2

There are a few other things to say about that code, and bear in mind, I'm only talking about my own code.

(1) I didn't realize it when I was writing Java/OOP code, but all of those methods that return `void` and take no input parameters have an impact on your brain and your thinking. You can't tell what the methods do by looking at their type signatures, so you either (a) trust that the methods are well-named, or (b) you have to look at their source code to see what they do.

As an example of what I mean, I trusted my own code because I knew that all of my getters and setters were just boilerplate code. But one time when I was debugging a problem with a junior developer, I found that he wrote a setter method like this:

```
public void setFoo(Foo newFoo) {
    openTheGarageDoor();
    walkTheDog();
    buyGroceries();
    solveWorldPeace();
    this.foo = newFoo + 1;
}
```

It wasn't quite that exhaustive, but there was a *lot* going on there. And the important part is that all of that happened inside a method with this type signature:

```
public void setFoo(Foo newFoo)
```

So, I learned some lessons:

- This developer needed more training
- I couldn't trust method type signatures, I had to look at their code

(2) A second point I'll add is that many Java methods can throw exceptions, and when they can, you have to think about (a) handling the successful case, and (b) handling the failure case, such as wrapping the method call with try/catch. This gets to be very hard on the brain, so I got to the point that — using the Model/View/Controller paradigm — I only handled exceptions with my top-level controllers. I logged exceptions where they occurred, but didn't catch them; I just let them bubble up to my controllers, and handled them there.

I didn't know anything about functional programming, and I couldn't think of a better way to deal with exceptions, so that's how I dealt with that problem.

2.0.3 Is Java/OOP code really that simple? A statistical look

I haven't worked with Java in quite a while now, so as I wrote this I wondered, "Was my code really that simple?"

To check what I was thinking, I searched three old Java codebases — all projects that were in production in their day. I just did a simple search for the `<.*>` pattern, and found these stats in those three projects:

1. 0.2%: 118 lines contain the `<.*>` pattern out of 74,200 lines
2. 1.4%: 686 lines out of 49,189
3. 1.2%: 313 out of 27,083

So out of about 150,000 lines of code there are 1,117 lines with a `<.*>` pattern, for a total of 0.74%. (Note that those line-count numbers include blanks lines and comment lines, so the actual percentages are higher.)

I then looked through all of those `<.*>` lines of output, and found that these were the two most difficult lines to read:

```
SortedMap<String, Integer> wordCountMap = new TreeMap();
public Class<?> getColumnClass(int columnIndex) {
```

So that's a look at some Java/OOP code. Next, let's look at some Scala code.

3
Beginning Scala programming

What happened to me when I started working with Scala can be summarized like this:

- I initially started using Scala as a "better Java"; I wrote OOP code with Scala
- As time went on I read Programming in Scala, and learned a little about the concept of a fusion of OOP and FP
- I started mixing pure functions into my OOP code
- I saw that pure functions were great because (a) output depended only on input, and (b) there were no side effects; they were super-easy to test
- I saw that classes like `Option`, `Try`, and `Either` were great alternatives to throwing exceptions; I no longer had to think about my code short-circuiting when something bad happened
- As I'll discuss shortly, because I knew exactly what was going into a function and coming out of a function, I began to trust type signatures again; I didn't have to read every function's code to see what it *really* did
- I realized that people were on to something: FP had real benefits, it wasn't a fad, and I wanted to learn more about it

Because so much of this has to do with *programming with types*, I'll start digging into types in the following discussion.

3.0.1 Pure functions

Pure functions are great because you can trust them.

Writing pure functions means:

- A function's output depends only on its input parameters
- The function does not modify its input parameters
- The function does not modify other "hidden" variables in its class

- The function does not communicate with the outside world: its has no I/O, no internet access, no database access
- Given the same input `X`, the function always returns the same output `Y`
- The function is total, meaning that it has a defined result for every possible input value

> Tip: As I describe in this article on partial functions, methods can also be *partial* rather than *total*.

Examples of pure functions include:

- Getting the length or checksum of a string
- Math functions like +, -, *, and /
- More math functions like sin, cos

A thing to note about pure functions is that they always take on or more input parameters, and they never return `Unit`:

```
def stringLength(s: String): Int
def cos(x: Double): Double
def sum(a: Int, b: Int): Int
def sum(nums: List[Long]): Long
```

Because pure function signatures always include the types of the input parameters, a cool thing is that you can often guess what a function does just by looking at its type signature, even if you make the function name meaningless. For example, as a little pop quiz, what do you think this function does:

```
def XYZ(s: String): Int = ???
```

Because the function takes a `String` input parameter and returns an `Int` — and also because you can be 100% sure that it's a pure function with no side effects — the function can only do a few things:

- Calculate the length of the string
- Calculate some sort of checksum of the string

I initially wrote that the function could attempt to convert the string to an integer, but that function would look like this:

```
def XYZ(s: String): Option[Int] = ???
```

So that's about it. Even without knowing the name of the function you can get a good idea of what it does just from reading its type signature. That's cool.

In fact, that's *very* cool, and even a game-changer. This one feature is one of the great things about functional programming:

> Because functions are pure, by just glancing at a function's type signature you can have a great feel for what it can (and can't) do.

Not to gush too much, but I love writing pure functions: it makes things *much* easier on my brain. When I'm writing a function I don't have to think about state of the entire application, I just have to think about (a) what's coming in, (b) my algorithm, and (c) what's going out.

And these days I also like reading the signatures of pure functions that other people write! Unlike OOP style *methods* that can return `void` and/or take no input parameters — which makes their type signatures meaningless — when I look at the type signature of a pure function, I can tell at a glance what's going on.

3.0.2 Another pop quiz

Here's another quiz: Knowing that this function is pure, what can it possible do:

```
def f(list: List[Int]): Int
```

As a followup to that question, here's the same function signature, but with a generic type; what can it possibly do:

```
def f(list: List[A]): A
```

(If you haven't seen this syntax before, `List[A]` means that the `List` can contain any type, such as a `List[Int]`, `List[String]`, `List[Person]`, etc.)

I won't give any answers for this one, but the important lesson here is that you can look

at the type signature of a pure function and generate some pretty good guesses about what that function does, even if the name of the function is meaningless.

4

Functional error handling in Scala

Here's an extremely important rule about pure functions:

> Pure function signatures don't lie.

What you see is what you get. And when it comes to handling exceptions that can occur in functions, well, this means that pure functions don't throw exceptions.

So what do they do? They return one of several different types. These are the error-handling types that are built into Scala:

- `Option`
- `Try`
- `Either`

There are other types in third-party libraries, but these are the built-in error handling types in the Scala library.

4.1 What this looks like

To demonstrate how this works, let's look at function that converts a `String` to an `Int`.

When you attempt to convert a string to an integer in Scala, you're primarily interested in converting a string like `"42"` into the integer `42`. We refer to this as the "happy path," because it's the success path for your function.

But you also have to account for the possibility of getting a string like `"foo"`, which doesn't convert to an integer. We refer to this as the "unhappy path," which a pure function also needs to account for.

And these two cases are exactly what Scala's error-handling types are created for. They let you encapsulate the potential happy and unhappy results within one type. Therefore, a pure `makeInt` function has type signatures like these:

```
def makeInt(s: String): Option[Int]
def makeInt(s: String): Try[Int]
def makeInt(s: String): Either[Throwable, Int]
```

Whoa. For a "simple" `String` to `Int` conversion function we're already seeing more complex types:

```
Option[Int]
Try[Int]
Either[Throwable, Int]
```

I'll come back to this in a few moments, but for the present moment, it's important to note that these types are already as difficult to understand as anything I ever wrote in Java. Because I'm just writing code as an A1 application programmer, this is significant.

Getting back to the point at hand … while these types require the brain to work a little harder, they're also cool because the function signature doesn't lie, and indeed, it tells you everything you need to know:

- The `makeInt` process can fail
- If it succeeds, the success information is inside those `Option`, `Try`, and `Either` types
- If it fails, the error information will also be contained in those types

So, once you know how `Option`, `Try`, and `Either` work, you're in great shape! Because pure function signatures don't lie, one look at these function signatures tells you that `makeInt` can fail.

Furthermore, because `Option`, `Try`, and `Either` work well in Scala for-expressions, you don't have to write a bunch of try/catch code to deal with these types. The for-expression handles both the happy and unhappy paths!

4.2 It's just algebra

It's important to mention that if this code looks scary:

```
def makeInt(s: String): Option[Int]
def makeInt(s: String): Try[Int]
def makeInt(s: String): Either[Throwable, Int]
```

it may help to know that it's just math, like algebra, high school algebra. Let's say you use that last function, the one written with `Either`. Depending on your preference, you can call it in two different ways:

```
val i = makeInt(s)                          // don't declare i's type
val i: Either[Throwable, Int] = makeInt(s)  // explicitly declare i's type
```

The first example shows how you call `makeInt` without declaring the type of `i`, and the second example shows how you declare `i`'s type explicitly. The two lines of code are equivalent.

Now, if you ignore types and classes completely for a few moments, let's just imagine that `makeInt` returns a two-element tuple:

```
val (e, i) = makeInt(s)
```

If you forget about the `val` keyword for a moment and further write that code like this:

```
(e, i) = makeInt(s)
```

you can see what I'm talking about: this is just algebra. The equation states that we're passing the parameter `s` into a pure function named `makeInt`, and `makeInt` returns two values, `e` and `i`, where `e` stands for the possible error that can occur, and `i` is the value you get when all goes as expected.

So if you like algebra, you're really going to like pure functions. With this approach, all of your code becomes a series of equations.

In fact, what happens is:

- You write all of your logic as a series of pure functions
- Then you combine those pure functions together, just like combining a series of

algebraic equations together

5
A "Word Occurrence" example

Writing Scala code like this gets to be so much like math that you can imagine walking in on Einstein writing at a chalkboard, and following his equations one at a time until he joins them all together at the end.

As an example, let's walk through the process of writing a "word occurrence" algorithm using only pure functions. I think you'll see that we can talk through this as we sketch out a series of function signatures, just like writing algebra on a chalkboard.

> This example was inspired by a Haskell book. I'd like to give it credit, but at the moment I can't remember which Haskell book I saw it in.

5.1 Problem statement

The first thing to do is to write a problem statement:

> Given a document that's represented as a `String`, count the number of occurrences of each word in that document.

When we're done, the output of our algorithm should look like this:

```
the    543
an     210
or     99
...
...
```

So, how do we start?

5.2 Getting started

Using functional programming and pure functions, the way I solve problems is to sketch function signatures. So in this case I know I want a word occurrences function:

```
def wordOccurrences(?): ?
```

Given the problem statement, I know it will take a `String` as input:

```
def wordOccurrences(s: String): ?
```

And given the example output, I know that the function's result should be stored in a map, specifically a map where the key is a string, and the value is an integer:

```
Map[String, Int]
```

Note that the output is printed in sorted order, so depending on the details of the requirements, you may also want to return a `LinkedHashMap` or `VectorMap` to retain the sorted order, and indicate to others that there's something unique about this map:

```
LinkedHashMap[String, Int]
```

For the purposes of this example I'll leave the result type as a `Map`, and let the calling program sort the map if they want, but you get the idea:

```
def wordOccurrences(s: String): Map[String, Int]
```

So there we have it, that's the signature for our main function. Now all we have to do is sketch out some more pure functions that this function will use.

5.3 Converting a document to words

The next thing I would think about is that inside this function, I have a document that's represented by a string, and I need to convert it into a series of words. So now I need a "convert document to words" function:

```
def convertDocumentToWords ...
```

Well, "a series of words" is just a sequence of strings, so I can sketch my function like this:

```
def convertDocumentToWords(): Seq[String] = ???
```

And what I'm calling a "document" as input is really just a string, so now I have this:

```
def convertDocumentToWords(s: String): Seq[String] = ???
```

Some programmers like to use aliases to make their code more readable, so using Scala 3 opaque types we can write this:

```
opaque type Document = String
opaque type Word = String
```

and then this:

```
def convertDocumentToWords(doc: Document): Seq[Word] = ???
```

Cool. That's a sketch of that function signature.

What do we need next? Well, now that we have a list of all of the words in the document, we need to convert that list into a `Map[Word, Int]`, i.e., a map of (a) each word that was found in the document to (b) the number of occurrences of that word.

> Note: I cover opaque types in detail in the Scala Cookbook.

5.4 Counting word occurrences

Following this function-sketching thought process, let's sketch another pure function. The previous function gives us a list of words that are in the document:

```
def convertDocumentToWords(doc: Document): Seq[Word] =
                                           ---------
```

So we can use that output as the input to our next function:

```
def countWordOccurrences(words: Seq[Word]) ...
                         ----------------
```

Because I stated above that the result we want from this function is a `Map[Word, Int]`, we can now completely sketch this function:

```
def countWordOccurrences(words: Seq[Word]): Map[Word, Int]
```

Given a list of words, this function returns a map of each word that was found, and the number of times it was found.

Hmmm, I think we're done … all we have to do is glue our pure functions together.

5.5 Combining our functions to create an application

Let's see if we can glue our pure functions together. Assuming that I use our opaque types, and shortening the main function name from `wordOccurrences` to `wc` (for "word count"), the main function signature looks like this:

```
def wc(d: Document): Map[Word, Int] = ???
```

Then, inside that function, the first thing we do is convert the document into a list of words:

```
def wc(d: Document): Map[Word, Int] =
    val listOfWords: Seq[Word] = convertDocumentToWords(d)
    ...
```

After that we convert the list of words into a map of word occurrences:

```
def wc(d: Document): Map[Word, Int] =
    val listOfWords: Seq[Word] = convertDocumentToWords(d)
    val wcMap: Map[Word, Int] = countWordOccurrences(listOfWords)
```

Then, ignoring sorting, we return that map:

```
def wc(d: Document): Map[Word, Int] =
    val listOfWords: Seq[Word] = convertDocumentToWords(d)
    val wcMap: Map[Word, Int] = countWordOccurrences(listOfWords)
    wcMap
```

If you prefer, you can reduce that code to this:

```
def wc(d: Document): Map[Word, Int] =
    val listOfWords: Seq[Word] = convertDocumentToWords(d)
```

```
    countWordOccurrences(listOfWords)
```

and then this:

```
def wc(d: Document): Map[Word, Int] =
    countWordOccurrences(convertDocumentToWords(d))
```

I often find that when I write my code with pure functions it ends up looking a little like Lisp, and that's true again here.

> As I mention in this article and this article, functional programmers use terms like "evaluation" and "substitution" to refer to reducing code like this.

That's pretty cool. We almost completely solved this problem by (a) sketching out a series of pure function signatures, and then (b) combining those signatures like a series of algebraic equations. All that's left to do is write the code inside those function bodies.

5.6 For the detail-oriented

You may have noticed that I skipped a few functions to get to that solution. For instance, `convertDocumentToWords` is going to need a few helper functions:

- Convert a document into paragraphs
- Convert a paragraph into words

If you follow the same thought process that I used above, you'll end up sketching a couple of pure function signatures like this:

```
splitDocumentIntoParagraphs(d: Document): Seq[Paragraph]
splitParagraphIntoWords(p: Paragraph): Seq[Word]
```

5.7 You're a translator

One thing to note in all of this is that you, the programmer, are a translator. When someone says "a series of words", you think:

```
Seq[String]
```

or

```
Seq[Word]
```

And when they say they need "a list of words and the number of times they occur," you think:

```
Map[String, Int]
```

or

```
Map[Word, Count]
```

Also, what's interesting about this process is that we're already working with (complicated) types like `Map[Word, Count]`, and we're creating aliases to make our types more meaningful. So a good question now is, does this seem hard, or does it seem relatively simple and logical?

5.8 Requirements Change: Read that document from a file

So far our solution looks like this:

```
def wc(d: Document): Map[Word, Count]
```

But what happens now if our requirement changes and we're told:

> You don't get a string for input, you have to read the document from a file.

My first thought is that we don't have to throw anything away. We already have this function that converts a string/document to the desired map, so all we need now is a wrapper function that reads a file into a string.

So let's sketch out this new function. Right away we know that we want to read a file:

```
def wc(f: File) ...
```

But what about our output?

Well, we know that we still want a `Map[Word, Count]` as a result, but when we're dealing

with a file, things can go wrong. For instance, the file might not exist or have the correct permissions.

In the object-oriented world I used to write code like this:

```
@throws FileNotFoundException
def wcFromFile(f: File) ...
```

But in FP we don't do that(!). We don't throw exceptions. Instead, we handle exceptional conditions using types.

In this example, because we're (a) using the standard tools that are built into Scala and (b) dealing with external resources like files, databases, and network I/O, we want to return an `Option`, `Try`, or `Either` type. More specifically, because we're working with files and we'll probably want access to the error message, we should use `Try` or `Either`. I'll use `Try`:

```
def wc(f: File): Try[?]
```

> If you're new to Scala, `Try` and `Either` will let you access the original exception, but `Option` does not.

This leads to the question, what goes inside that `Try` declaration?

The answer is that the function's result goes inside `Try`. Because the function returns a `Map[Word, Count]` when it succeeds, the function's return type now looks like this:

```
def wc(f: File): Try[Map[Word, Count]]
```

When it's run, this function returns one of two possible subclasses of `Try`:

- `Success`
- `Failure`

For example, when your function succeeds, it returns this type:

```
Success[Map[Word, Count]]
```

But if the function can't read the file, it returns a `Failure`, and that `Failure` object contains the failure reason.

One way that the code that calls the `wc` function can handle the `Try` result is like this:

```
// this is code that calls our new 'wc' function
wc("/Users/alvin/my-file.txt") match
    case Success(map) => // use the map here
    case Failure(msg) => // handle the error message here
```

> For more information on how this works, see my article, Functional error handling in Scala

So, in summary, when you're told that you have to read the document from a file, this is our solution:

```
def wc(f: File): Try[Map[Word, Count]]
```

If you prefer `Either`, the solution looks like this:

```
def wc(f: File): Either[Throwable, Map[Word, Count]]
```

This code states that if everything goes fine, you'll get a `Map[Word, Count]` in the `Either`, but if something fails, you'll get a `Throwable`.

> For more information on how to use `Either`, see my article, A Scala Either, Left, and Right example.

6
Thinking With Types: Summary

Earlier in this booklet I showed that these were the two most complicated type declarations I found in over 150,000 lines of production Java code:

```
SortedMap<String, Integer> wordCountMap = new TreeMap();
public Class<?> getColumnClass(int columnIndex) {
```

Conversely, as you saw in the previous example, I just sketched out a little "Word Occurrences" application in a Scala/FP style, and while writing less than ten lines of code I already used types like these:

```
Try[Map[Word, Count]]
Either[Throwable, Map[Word, Count]]
```

So the first obvious thing to say is that in Scala/FP we use types a lot more than developers are used to using in Java/OOP.

Because I'm interested in helping programmers who are interested in making the conversion from Java to Scala, it's important to acknowledge up front that this is a different world than Java/OOP. In the Java world you generally only work with types like this if you're a library developer, but in the Scala/FP world, developers at all levels work with them.

That being said, let me ask this: Did you find the Word Occurrences example hard to follow? I suspect — and hope — it wasn't hard to follow. I just sketched out a series of pure functions and their type signatures. Personally, I think it's very cool that you can sketch a solution like this by just thinking about the types.

> It's important to note that if you just want to use Scala as a "Better Java" and write Scala/OOP code, you can do that as well. In this case, the types you see won't be any more difficult than what you see in Java/OOP.

6.1 Benefits

There are a lot of benefits of "Thinking With Types." First, here's some of what you've already seen:

- You write pure functions
- Writing pure functions is simpler than impure functions; you don't have to worry about the state of the entire application, all you have to think about is (a) the data that comes in, (b) your algorithm, and (c) the data that goes out
- Following the same logic about global state, *testing* pure functions is also simpler
- You can trust pure functions, they do not lie: their type signatures tell you what's going on
- When you look at a pure function's signature and see `Option`, `Try`, or `Either`, you immediately know, "Something can go wrong here"

Beyond those benefits:

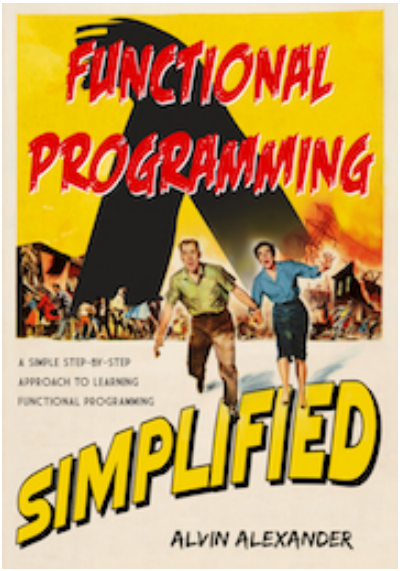
- You'll start to see solving problems as being like algebra or some other form of math
- As I did with the Word Occurrences example, as you work on solving a problem, you'll sketch out a series of function signatures; the details are important later, but you can solve problems at a high level just by sketching function signatures and working with the types
- As you also saw with that example, the final solution to the problem is just a matter of combining your pure functions

6.1.1 Functional Programming

As you may have guessed, Thinking With Types is related to functional programming. While using as many pure functions as you can is a great step to improving your code, this thought process also provides a gateway to FP: If you like the style of programming you saw with the Word Occurrences example, you'll like FP.

6.2 For more information

In summary, I hope this brief booklet has helpful. I intentionally want to keep this short, but if you're interested in more details on these topics, see my books:



- Scala Cookbook, 2nd Edition
- Functional Programming, Simplified

All the best,
Al
alvinalexander.com