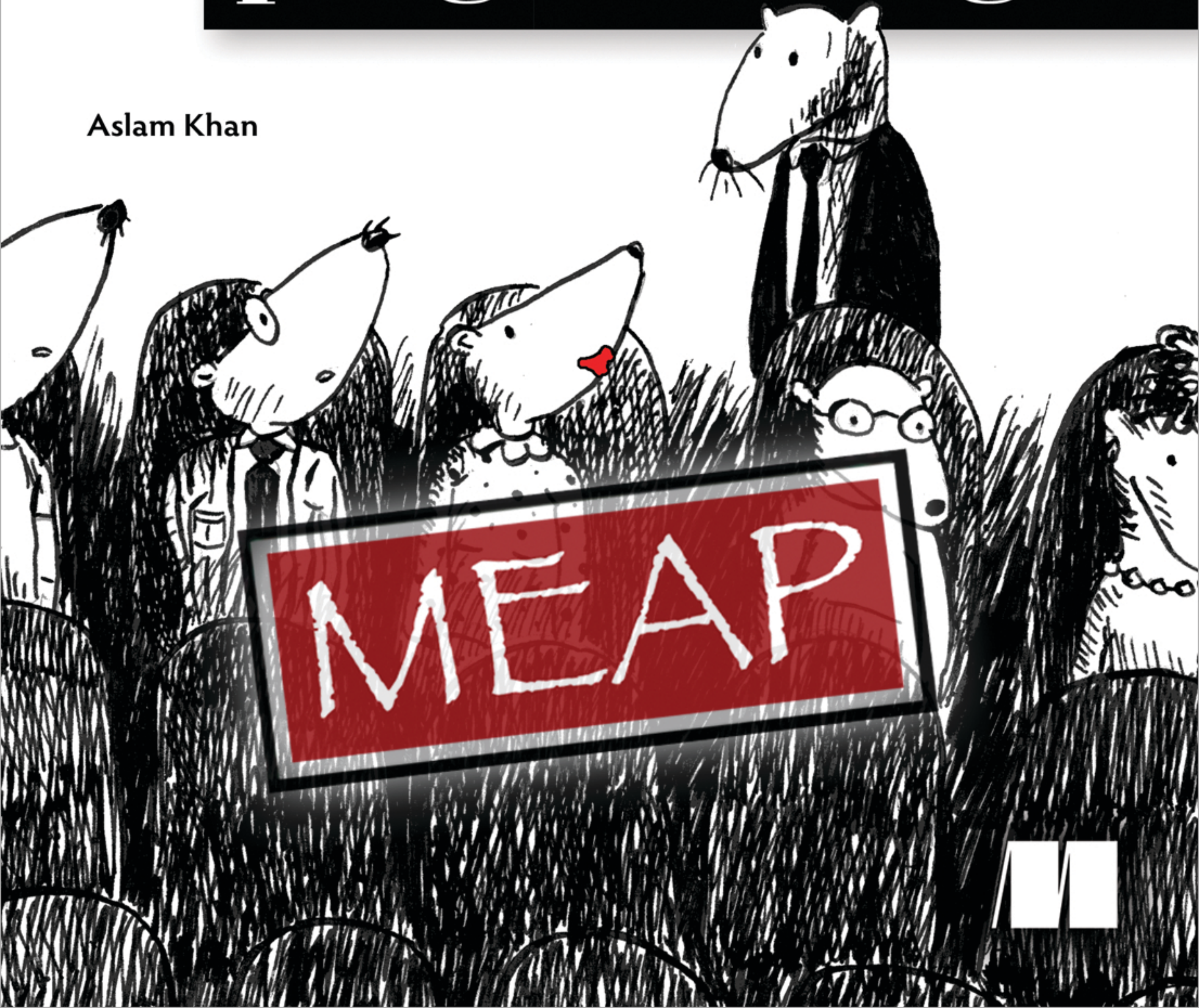
grokking functional programming

Aslam Khan



MEAP Edition
Manning Early Access Program
Grokking Functional Programming
Version 5

welcome

Thank you for purchasing the MEAP for *Grokking Functional Programming*. Many of us have attempted learning FP and found it to be far from simple to learn. It is easy to get lost in the maths and theory of computer science. I had many false starts learning FP too—the more I got into it, the more daunting it seemed. From that experience, I set a very lofty goal of making functional programming accessible and practicable in our everyday programming tasks.

We assume that you know some object oriented programming but do not expect you to be an OOP grand master. We use Java as the contrasting OOP language and Scala as the FP language. But this is not a Scala book. Nor do you need to know any Scala to use this book. In fact, you don't even need to know Java all that much either.

This book is tuned to integrate well with how people learn. We have been extremely particular about not leaving you on the edge of mental cliffs looking across a gaping chasm to the next concept. But this is not a perfect book and there may well be a few mental cliffs. We want to know about those. Whenever you find yourself confused or the pace too fast, or discover gaps in your understanding, please share those with us on the author forum. With your feedback, we might just fulfill our crazy objective.

To start off with, we are releasing the first three chapters. The first chapter sets the scene for the book, how to use it, what it expects of you and what you can expect of the book. By the end of this chapter, you will have Scala installed on your computer and successfully executed a few simple FP code snippets. There is nothing daunting here.

In chapter 2, we lead you through a set of familiar code examples to the fundamental core of FP. This is the idea of referential transparency. We believe that this is the heart of FP and all other FP concepts exist because of our wish for referential transparency. This might cause a bit of a stir in the theoretical circles of FP. That's ok. Our quest is to teach FP in the simplest possible way, not attaining theoretical perfection. Mostly in this chapter, you will refactor snippets of object oriented code towards an increasingly FP style. At the end, we share why everything else hangs off referential transparency.

Chapter 3 is where we dabble with the first weird aspect of FP—programming with data that can never be modified. This notion of immutable data goes against everything that you probably learned on day 1 of programming. This is a big topic and it is covered from different perspectives in future chapters. It is actually a constant thread throughout the book. For this chapter, we focus on iterating or looping through collections. Some interesting concepts such as recursion emerge. And you get to write your first higher order function too.

Looking ahead, we will spend more time on programming with immutable data. This will lead us to the elephant in the room—does FP really make concurrent programming any easier? Along the way, we will introduce more about higher order functions, functional data structures and other FP concepts. More often than not, we will expand on these concepts just when we need it.

Enjoy these first few chapters of our book. Do share your points of confusion and other suggestions on the Author forums. Together, we will produce a book on functional programming for the rest of us. And most of all, enjoy the journey into FP land.

—Aslam Khan

brief contents

learning functional programming 1

Why are we here? & What are we going to do?

“Yes, there are two paths you can go by, but in the long run
There's still time to change the road you're on.”
-- Led Zeppelin, Stairway to Heaven

Perhaps you picked up this book because...

1 **You are curious about functional programming.** You heard about functional programming, read the Wikipedia entry, and looked at a few books too. Maybe you rolled your eyes at the maths behind all the code. Yet it still holds a curiosity in you.

(In our hearts, we set out to write the least intimidating functional programming book ever. That is something that you will judge, not us.)

2 **You tried to learn functional programming before.** Many of us have tried learning functional programming more than once, and still don't get it. Just when we understand one key concept, the next recedes away even faster. It is learning by chasing shadows. Not fun at all.

{We believe it does not have to be like this. Learning functional programming should be enjoyable. We want your endorphins to propel you to the next page.}

3 **For you, the jury is still out.** The majority of us have been programming for many years in an object oriented programming language like C#, Java or Ruby. We have heard the buzz of FP, read some blog entries, and tried a bit of code. Still, we cannot see how it makes our programming life better.

[Yes, we also heard the multi-core, concurrency promises. But we want to go beyond that to help you include functional programming in your mental toolbox.]

Whatever your reason, we want you to...

- learn through experimentation and play
- never fear asking a silly question
- feel yourself growing to new levels

Don’t read this book if...

You are an expert functional programmer. Perhaps you have been programming in Scala, Clojure, F#, Haskell and other functional languages for several years. At best, you will find this book a curiosity in the way we teach functional programming.

or

You need to know the maths. Functional programming is built upon very rigorous mathematics. If you want to gain a deep understanding of lambda calculus, category theory and related mathematics, then you will need to look elsewhere. This book has absolutely none of the maths and theory that forms the bedrock of functional programming.

but...

What if you believe that functional programming cannot be taught without the maths?

This book exists because we believe that many developers will appreciate functional programming through practice more than through theory. Certainly, the decision is yours to make: give the book a chance or give it a miss. The comforting thought is this is not a life or death decision. It’s ok to change your mind, anytime.

What do you need to know before we start?

We assume that you have been developing software in any of the popular languages such as Java, C#, Ruby or Python. If you can satisfy yourself with this quick checklist, then you should be able to follow along comfortably.

You need to ...

- be familiar with object orientatation concepts like classes, objects and inheritance.
- be able to read Java code. . If you can understand the following Java code, you will be able to follow along easily.

Book has a title and a list of Author objects

Constructor: Creates a new Book object with a title and no authors.

add an Author for this book.

```
class Book {
  private String title;
  private List<Author> authors;

  public Book(String title) {
    this.title = title;
    this.authors = new List<Author>();
  }

  public void addAuthor(Author author) {
    this.authors.add(author);
  }
  ...
}
```

You don't need to be ...

- be an expert in object orientation
- be a Java master
- know anything about any functional programming languages such as Scala, F#, Clojure, Haskell and others.

What language are we going to use?

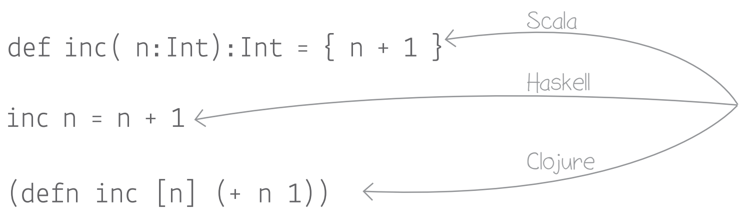
The majority of the examples and exercises use **Scala** as the programming language. We use Scala to explain functional programming concepts. Many of the examples can be written in more idiomatic Scala. The examples are deliberately written to highlight functional programming concepts in favour of "better" Scala.

WARNING: This is not a Scala language book but you will learn enough Scala as a wonderful side effect of learning functional programming.

At times, we may use another language because that language will illustrate a particular concept best. Sometimes, comparing the same code written in other languages also helps to emphasise a concept.

Remember, our focus is on functional programming. We know we have understood functional programming when we can see the differences in language features that support functional programming concepts as first class citizens. This is important because it gives us the knowledge to make pragmatic decisions in code - when to lean on the compiler, or when to implement by hand.

Don't get scared by syntax



```
def inc( n:Int):Int = { n + 1 }

inc n = n + 1

(defn inc [n] (+ n 1))
```

Functional programming languages can have really strange syntax. Here are functions that add 1 to a number.

What this book expects of you

To get the most out of this book may require an adjustment to the way most software development books are read. Do not expect to read from cover to cover, like a bedside read.

Instead, keep the book at your desk, working with it next to your keyboard and a few sheets of blank paper to write on. Shift perspective from being a receiver of thoughts to being an active participant.

Some tips for learning with this book

1 **Do the exercises.** Make a commitment to do every exercise. Feel the code as you type it, letting your subconscious mind absord what you do physically. Resist the temptation to cut 'n paste code.

2 **Create a space to learn.** Keep paper nearby and a few pencils or pens of different color. Sharpies or flipchart markers are good too. We want to create a space that radiates information, not a dull, sterile place.

3 **Get up and move around.** Write code by hand on a vertical surface then stand back and look at your code from a distance. Apart from the physical movement, we are changing perspectives too.

4 **Don't rush.** Work at a pace that is comfortable. It's ok to not have a consistently steady pace too. Sometimes we run, and other times we crawl. Sometimes, we do nothing. Rest is important so don't ignore fatigue.

What to expect from this book

We want to have a fair chance of learning functional programming. We don't want to get bogged down with theory and examples from the world of mathematics.

We also want to enjoy learning functional programming. We should not feel frustrated, confused and irritated every step of the way.

> “x.x represents the identity function, x.x, and (x.x)y represents the identity function applied to y. Further,(x.y) represents...
>
> (y.t)[x:=r] = y.(t[x:=r]) if x≠y and y is not in the free variables of r ...”

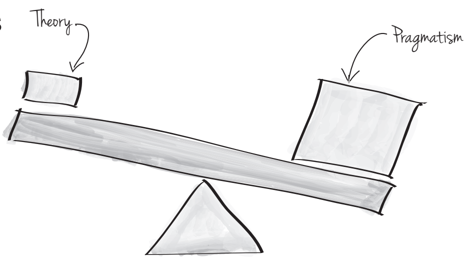
lambda calculus

The maths behind functional programming can leave us feeling defeated, when all we want to know is how to practically use functional programming in our day to day work.

We believe that it is possible to be a **pragmatic functional programmer** without being a master of the underlying mathematical theory.

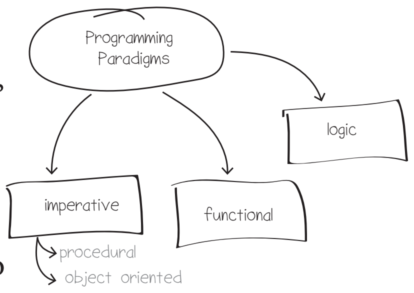
At the risk of being flamed

We will deliberately err on the side of practical usage as opposed to theoretical correctness. We will explain concepts and techniques in a way that may incite functional programming flame wars. This is a deliberate learning strategy to keep us moving forward as opposed to stalling at one theoretical concept. There will be a time that warrants theoretical perfection. For now, we shall focus a practical appreciation.



Where does functional programming fit?

There are three programming paradigms - logic, functional and imperative. Without doubt, imperative programming, is the most popular. In this book we will focus on crossing over from imperative programming to functional programming.



Within imperative programming, we have procedural programming with languages like C. Alongside that is object oriented programming (OOP) with languages like C++, Java and C#.

In the last twenty years and maybe a bit more, we have successfully built quite sophisticated systems using an imperative paradigm. The current popularity of OOP, in particular, cannot be disputed.

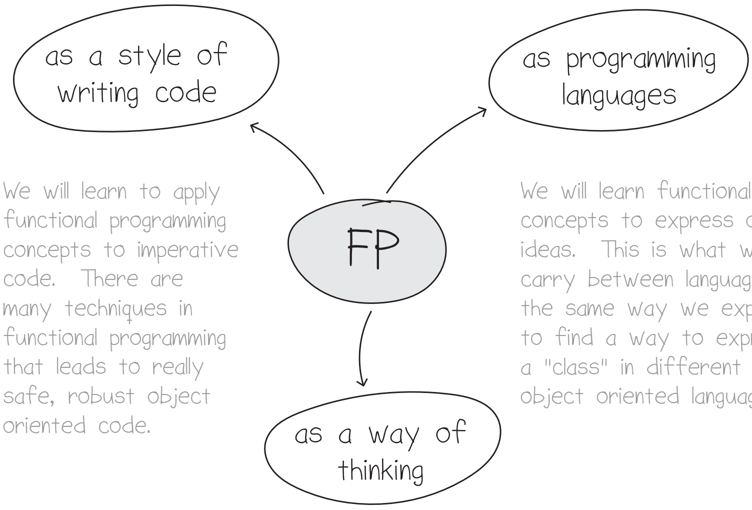
"Why change from the successful OOP to FP?"

That's a tough question. Perhaps, FP is superior to OOP. Maybe, FP is just different from OOP and, yet, equally powerful. There's also a chance that FP is better for a certain class of problems.

Whatever, the reasons may be, we believe that there is place for FP and OOP. It is an advantage to have more than one way of thinking in our mental toolbox. We certainly want to improve our chances of analysing problems and designing solutions. That's an advantage we should not disregard.

Not just programming in a new language

There are three perspectives that we carry through the book



We will learn to apply functional programming concepts to imperative code. There are many techniques in functional programming that leads to really safe, robust object oriented code.

We will learn functional concepts to express our ideas. This is what we carry between languages; the same way we expect to find a way to express a "class" in different object oriented languages.

This is the most difficult skill to master. It is training our mind to analyze problems differently and to design solutions differently too. This interplays with "as a programming language" and it needs a lot of practice.

In this book, we take a balanced view.
We acknowledge that FP and OOP are different. Each have different values, and there are some values that are common too. We embrace these differences as just that.

Get our tools ready

Now that we understand what to expect of this book, and what is expected of ourselves, it is time to get the last bit of preparation out of the way. We must install Scala and ensure that it works.

1 **Install Java.** Download Java from http://java.com/en/download/index.jsp. Make certain that you use Java 6 or later. Ensure that you have the environment variable `JAVA_HOME` set to the directory of your Java installation, and that the `bin` folder is in your path.

2 **Install Scala.** Download an installer for your operating system or a package of the Scala binaries from http://www.scala-lang.org/downloads. Ensure that you install version 2.10 or later. Then add the Scala `bin` folder to your shell's path.

3 **Run it!** In your shell, run the command `scala`. This will start the Scala REPL.

What is a REPL?
REPL stands for ***Read Evaluate Print Loop***. It reads a line of code that you type, evaluates that line of code and prints the result to the screen, and loops, waiting to read the next line.

your version of scala may be different

```
Welcome to Scala version 2.10.1 ...
Type in expressions to have them evaluated.
Type :help for more information
scala>
```

This is the scala prompt where we enter commands and code.

Try it! In your Scala REPL, try the following.

```
scala> print "Hello, World"
Hello World

scala> var n = 20
n: Int = 20

scala> n * 2 + 2
res0: Int = 42
```

Type this and hit Enter

This is the output

All scala variables are preceded with `var`

The REPL created `n` as an `Int` with value `20`

To quit the REPL and return to the shell, enter the command `:exit`.

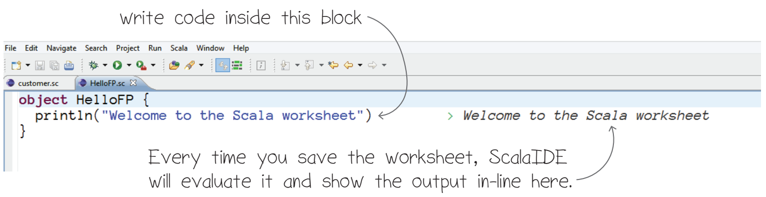
`res0` is a variable that the REPL created. It is of type `Int`, and has the value `42`. Try this: `res0/2`. The output should be `21`.

What tools can we use?

Can we use an IDE?

There is an open source pre-configured version of Eclipse for Scala known as the ScalaIDE. It can be downloaded from http://scala-ide.org/download/current.html.

ScalaIDE has a feature that acts as a REPL. To use it, create a Scala project. Then create a new "Scala Worksheet" and give it a name, for example `HelloFP`. It will present you with a code editor with a block of code similar to this.


write code inside this block
File Edit Navigate Search Project Run Scala Window Help
customer.sc
HelloFP.sc
Every time you save the worksheet, ScalaIDE will evaluate it and show the output in-line here.

```
object HelloFP {
  println("Welcome to the Scala worksheet")      > Welcome to the Scala worksheet
}
```

Can we use our favorite editor?

Certainly, we can use our favorite editor. There is syntax highlighting support and REPL plugins for many of the popular editors such as vi, emacs, Sublime, TextMate and others.

It is better to start of with simple tools that are comfortable enough. As we get more familiar with FP, we can start tweaking our tooling to support our deeper knowledge. For this book, an editor with syntax highlighting and a console REPL is sufficient.

Coffee Break

Make a cup of your favorite hot drink. While you're having a sip, try to figure out what each line of code does.

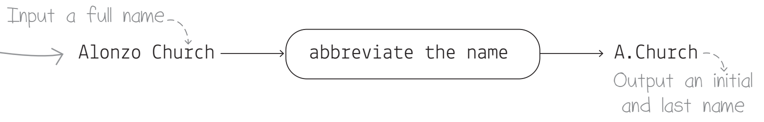
- `(1 to 10)`
- `(1 to 10).sum`

Jot down your answer on a page, and double check it in the REPL.

A little functional fun

Our first experiment is to abbreviate the name of a person into an initial and last name. Fire up the Scala REPL, and code along.

Alonzo Church was a mathetician who developed lambda-calculus, the foundation of FP.



1

```
scala> var name = “Alonzo Church”
name: String = Alonzo Church
```

We never specified the type of name. Scala evaluated the expression, saw it was a String and inferred that name can only be a String.

Split name at the embedded space and store the result in names.

2

```
scala> var names = name.split(“ “)
names: Array[String] = Array(Alonzo, Church)
```

names is an Array of Strings

it is has two elements - Alonzo and Church

Arrays are zero-indexed and accessed using ( ) as opposed to [ ] in many other languages. This picks out the first element of names being Alonzo.

Use the function take to extract the first element from the string "Alonzo" which is "A"

3

```
scala> var initial = names(0).take(1)
initial: String = “A”
```

Using take(1), is the same as applying a substring function from position 0 for a length of 1.

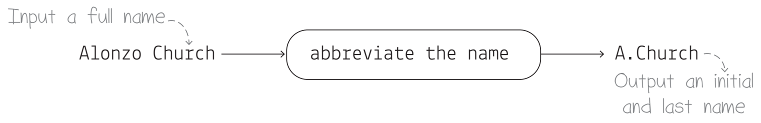
Use + to concatenate initial with "." and the second element of the names array.

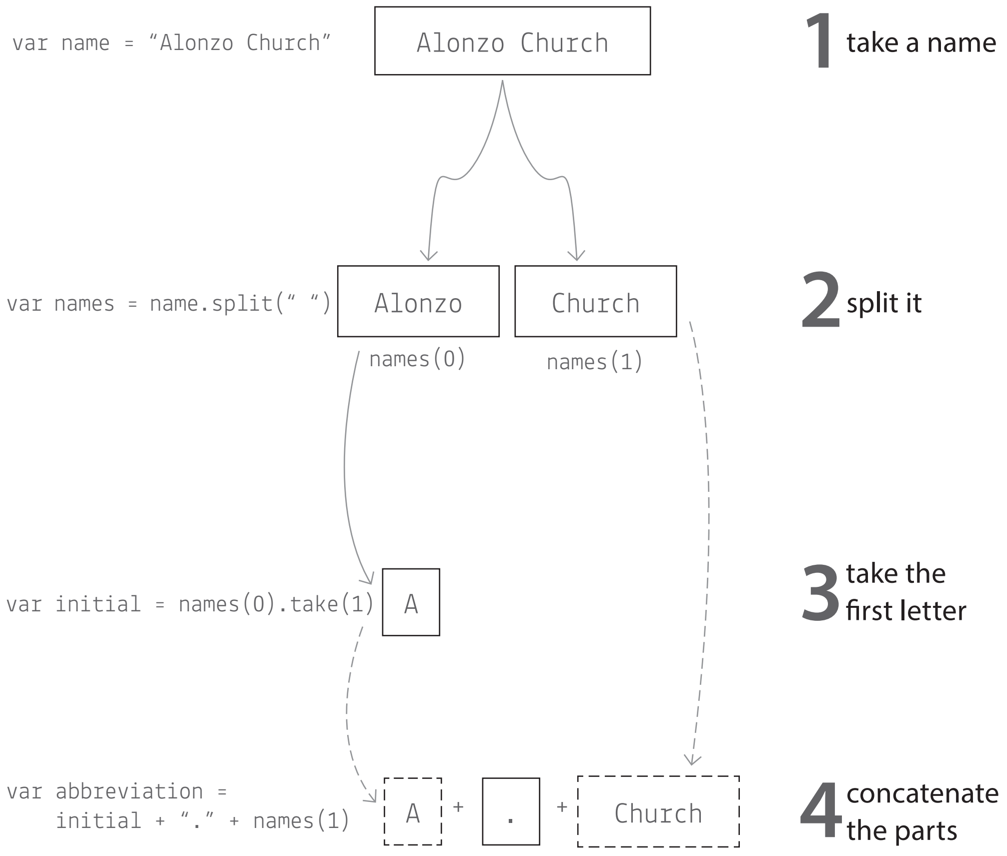
4

```
scala> var abbreviation = intial + “.” + names(1)
abbreviation: String = “A.Church”
```

Abbreviating names explained

What we just did to transform a name into its initial and last name is exactly how we would abbreviate the name if we were working without a computer.





```
var name = "Alonzo Church"
var names = name.split(" ")
var initial = names(0).take(1)
var abbreviation =
    initial + "." + names(1)
```

Coffee Break exercises explained

What do these lines of code do?

- `(1 to 10)`
- `(1 to 10).sum`

Let's try them in the REPL.

```
scala> (1 to 10)
res0: scala...Range = Range(1,2,3,4,5,6,7,8,9,10)
```

It creates a range from 1 to 10. There is syntactic sugar at work here. The compiler inserts a few "hidden" bindings so that it reads like `1.to(10)`. In other words, `to` is a method on the `1` object of type `Int` that returns a `Range` object. We can imagine the method signature as `Range to(Int i)` on the class `Int`.

A difference from Java is that Scala numeric literals are real objects, not primitives. Try this in the REPL -
`1.toString()`

```
scala> (1 to 10).sum
res1: Int = 55
```

The equivalent code in Java would be -

```
int range[] = {1,2,3...10}
int total = 0;
for (int i : range) {
  total = total + i;
}
```

Now that we know that `(1 to 10)` creates a `Range` object, then sum adds up the values in the `Range` object. In this case the sum of all numbers from `1` to `10` is `55`. Without syntactic sugar, we can rewrite the expression as `1.to(10).sum`. We can imagine the signature of the `sum` method on the `Range` object as `Int sum()`.

This is why functional programming is often described as "declarative programming." Even if we don't use a functional language, being declarative will yield better interfaces in OOP.

When programming the Java version, we express how we want to calculate the sum of the range. In the Scala version, we express what we want done with the range. In functional programming, we tend to think about what we want to achieve. We just declare our intentions in functional programming compared to step-wise instructions of how to achieve our intention.

More fun with Strings

Let us experiment a bit more with Strings. Work on a page first, just writing down what you imagine the code should be and the approximate output. Then open the REPL and confirm your thoughts.

This is just like a real experiment: make a hypothesis and prove or disprove it with practical observation. The difference from a science experiment is that it is often fun and rewarding to go off on little tangents too.

String manipulation

For the following REPL session, work out what the ??? should be. Remember to work on paper, then verify in the REPL.

```
scala> var name = "A Church"
name: String = A Church

scala> name.drop(2)
res0: String = ???

scala> name
name: String = ???

scala> res0.take(1)
res1: String = ???

scala> scala> ???
res2: String = AC

scala> name
name: String = ???

scala> ???
res4: String = CA
```

In your REPL, it might not be res0 because you may have been evaluating many expressions. To get it to be res0, restart the REPL.

HINT: use the reverse function on the String object. For example: "FP".reverse yields "PF".

Fun with Strings explained

Here are the solutions for the previous set of experiments. Again, follow along in the REPL to experience it first hand.

```
scala> var name = “A Church”
name: String = A Church
```

the function drop(n) literally drops n characters from the beginning of a String.

```
scala> name.drop(2)
res0: String = Church

scala> name
name: String = A Church
```

Just as in Java, Strings are immutable. Here, the drop function, leaves name unchanged.

Take the 1st character.

```
scala> res0.take(1)
res1: String = C

scala> name.take(1) + res1
res2: String = AC
```

Concatenate the 1st and 3rd characters

```
scala> name
name: String = A Church
```

Don’t forget, Strings are immutable

```
scala> res2.reverse
res3: String = CA
```

The full expression is
`(name.take(1) + name.drop(2).take(1)).reverse`

Summary

We established that this book will focus on teaching functional programming pragmatically as opposed to theoretically. We will not focus on mathematics of functional programming. We need to be familiar with object oriented programming but we don't need to be expert object oriented programmers.

In order to gain value from this book, we should be prepared to practice and work through the exercises.

Our first taste of functional programming was using the Scala REPL to do a few simple string manipulations using a handful of novel String functions.

We also touched on the notion of declarative programming as a style of functional programming. A declarative programming style emphasises expressing what we want and not how to achieve it.