FUNCTIONAL PROGRAMMING

A SIMPLE STEP-BY-STEP
APPROACH TO LEARNING
FUNCTIONAL PROGRAMMING

FREE PREVIEW!

SIMPLIFIED

ALVIN ALEXANDER

Functional Programming, Simplified

(Scala edition)

Alvin Alexander

Copyright

Functional Programming, Simplified

Copyright 2017 Alvin J. Alexander

All rights reserved. No part of this book may be reproduced without prior written permission from the author.

This book is presented solely for educational purposes. While best efforts have been made to prepare this book, the author makes no representations or warranties of any kind and assumes no liabilities of any kind with respect to the accuracy or completeness of the contents, and specifically disclaims any implied warranties of merchantability or fitness of use for a particular purpose. The author shall not be held liable or responsible to any person or entity with respect to any loss or incidental or consequential damages caused, or alleged to have been caused, directly or indirectly, by the information or programs contained herein. Any use of this information is at your own risk.

Version 1.0, published December 7, 2017

Contents

1 Introduction
(or, Why I Wrote This Book) 1

2 Who This Book is For 5

3 Goals, Part 1: “Soft” Goals of This Book 9

4 Goals, Part 2: Concrete Goals 15

5 Goals, Part 3: A Disclaimer 17

6 Question Everything 19

7 Rules for Programming in this Book 27

8 One Rule for Reading this Book 31

9 What is “Functional Programming”? 33

10 What is This Lambda You Speak Of? 43

11 The Benefits of Functional Programming 59

12 Disadvantages of Functional Programming 79

13 The “Great FP Terminology Barrier” 99

14 Pure Functions 107

15 Grandma’s Cookies (and Pure Functions) 115

16 Benefits of Pure Functions 125

17 Pure Functions and I/O 133

18 Pure Function Signatures Tell All 141

19 Functional Programming as Algebra 147

20 A Note About Expression-Oriented Programming 161

21 Functional Programming is Like Unix Pipelines 165

22 Functions Are Variables, Too 183

23 Using Methods As If They Were Functions 199

24 How to Write Functions That Take Functions as Input Parameters 211

25 How to Write a ‘map’ Function 231

26 How to Use By-Name Parameters 237

27 Functions Can Have Multiple Parameter Groups 251

28 Partially-Applied Functions (and Currying) 273

29 Recursion: Introduction 289

30 Recursion: Motivation 291

31 Recursion: Let’s Look at Lists 295

32 Recursion: How to Write a ‘sum’ Function 303

33 Recursion: How Recursive Function Calls Work 311

34 Visualizing the Recursive sum Function 317

35 Recursion: A Conversation Between Two Developers 325

36 Recursion: Thinking Recursively 329

37 JVM Stacks and Stack Frames 337

38 A Visual Look at Stacks and Frames 345

39 Tail-Recursive Algorithms 353

40 A First Look at "State" 365

41 A Functional Game (With a Little Bit of State) 369

42 A Quick Review of Case Classes 385

43 Update as You Copy, Don't Mutate 393

44 A Quick Review of `for` Expressions 403

45 How to Write a Class That Can Be Used in a `for` Expression 405

46 Creating a Sequence Class to be Used in a `for` Comprehension 407

47 Making `Sequence` Work in a Simple `for` Loop 409

48 How To Make `Sequence` Work as a Single Generator in a `for` Expression 411

49 Enabling Filtering in a `for` Expression 413

50 How to Enable the Use of Multiple Generators in a `for` Expression 415

51 A Summary of the `for` Expression Lessons 417

52 Pure Functions Tell No Lies 419

53 Functional Error Handling (Option, Try, or Either) 421

54 Embrace The Idioms! 423

55 What to Think When You See That Opening Curly Brace 425

56 A Quick Review of How `flatMap` Works 427

57 `Option` Naturally Leads to `flatMap` 429

58 `flatMap` Naturally Leads to `for` 431

59 `for` Expressions are Better Than `getOrElse` 433

60 Recap: `Option` -> `flatMap` -> `for` 435

61 A Note About Things That Can Be Mapped-Over 437

62 A Quick Review of Companion Objects and `apply` 439

63 Starting to Glue Functions Together 441

64 The “Bind” Concept 443

65 Getting Close to Using `bind` in `for` Expressions 445

66 Using a “Wrapper” Class in a `for` Expression 447

67 Making `Wrapper` More Generic 449

68 Changing "new Wrapper" to "Wrapper" 451

69 Using `bind` in a `for` Expression 453

70 How `Debuggable`, `f`, `g`, and `h` Work 455

71 A Generic Version of `Debuggable` 457

72 One Last `Debuggable`: Using `List` Instead of `String` 459

73 Key Points About Monads 461

74 Signpost: Where We're Going Next 463

75 Introduction: The `IO` Monad 465

76 How to Use an `IO` Monad 467

77 Assigning a `for` Expression to a Function 469

78 The `IO` Monad and a `for` Expression That Uses Recursion 471

79 Diving Deeper Into the `IO` Monad 473

80 I'll Come Back to the `IO` Monad 475

81 Functional Composition 477

82 An Introduction to Handling State 479

83 Handling State Manually 481

84 Getting State Working in a `for` Expression 483

85 Handling My Golfing State with a State Monad 485

86 The `State` Monad Source Code 487

87 Signpost: Getting IO and State Working Together 489

88 Trying to Write a `for` Expression with IO and State 491

89 Seeing the Problem: Trying to Use State and IO Together 493

90 Solving the Problem with Monad Transformers 495

91 Beginning the Process of Understanding `StateT` 497

92 Getting Started: We’re Going to Need a `Monad` Trait 499

93 Now We Can Create StateT 501

94 Using StateT in a `for` Expression 503

95 Trying to Combine `IO` and `StateT` in a `for` Expression 505

96 Fixing the `IO` Functions With Monadic Lifting 507

97 A First IO/StateT `for` Expression 509

98 The Final IO/StateT `for` Expression 511

99 Summary of the `StateT` Lessons 513

100Signpost: Modeling the world with Scala/FP 515

101What is a Domain Model? 517

102A Review of OOP Data Modeling 519

103Modeling the “Data” Portion of the Pizza POS System with Scala/FP 521

104First Attempts to Organize Pure Functions 523

105Implementing FP Behavior with Modules 525

106Implementing the Pizza POS System Using a Modular Approach 527

107The “Functional Objects” Approach 529

108Demonstrating the “Functional Objects” Approach 531

109Summary of the Domain Modeling Approaches 533

110ScalaCheck 1: Introduction 535

111ScalaCheck 2: A More-Complicated Example 537

112The Problem with the IO Monad 539

113Signpost: Type Classes 541

114Type Classes 101: Introduction 543

115Type Classes 102: The Pizza Class 545

116Type Classes 103: The Cats Library 547

117 Lenses, to Simplify "Update as You Copy" 549

118 Signpost: Concurrency 551

119 Concurrency and Mutability Don't Mix 553

120 Scala Concurrency Tools 555

121 Akka Actors 557

122 Akka Actor Examples 559

123 Scala Futures 561

124 A Second Futures Example 563

125 Key Points About Futures 565

126 A Few Notes About Real World Functional Programming 567

127 Signpost: Wrapping Things Up 569

128 The Learning Path 571

129 Final Summary 573

130 Where To Go From Here 575

A Explaining Scala's `val` Function Syntax 579

B The Differences Between `val` and `def` When Creating Functions 581

C A Review of Anonymous Functions 583

D Recursion is Great, But … 585

E `for` expression translation examples **587**

F On Using `def` vs `val` To Define Abstract Members in Traits **589**

G Algebraic Data Types **591**

1

Introduction (or, Why I Wrote This Book)

> *"So why do I write, torturing myself to put it down? Because in spite of myself I've learned some things."*
>
> Ralph Ellison

The short version of "Why I wrote this book" is that I found that trying to learn functional programming in Scala was really hard, and I want to try to improve that situation.

The longer answer goes like this …

My programming background

My degree is in aerospace engineering, so the only programming class I took in college was a FORTRAN class I was forced to take. After college I was one of the youngest people at the aerospace company I worked at, which meant that I'd have to maintain the software applications our group used. As a result, I became interested in programming, and then became interested in (a) "How can I write code faster?", and then (b) "How can I write maintainable code?"

After that I taught myself how to program in C by reading the classic book, The C Programming Language by Kernighan and Ritchie, quickly followed by learning Object-Oriented Programming (OOP) with C++ and Java. That was followed by investigating other programming languages, including Perl, PHP, Ruby, Python, and more.

Despite having exposure to all of these languages, I didn't know anything about Functional Programming (FP) until I came across Google's Guava project, which includes FP libraries for Java collections. Then, when I learned Scala and came to understand the methods in the Scala collections' classes, I saw that immutable values and pure functions had some really nice benefits, so I set out to learn more about this thing called *Functional Programming*.

Trying to learn FP with Scala

As I tried to learn about FP in Scala, I found that there weren't any FP books or blogs that I liked — certainly nothing that catered to my "I've never heard of FP until recently" background. Everything I read was either (a) dry and theoretical, or (b) quickly jumped into topics I couldn't understand. It seemed like people enjoyed writing words "monad" and "functor" and then watching me break out in a cold sweat.

As I googled "scala fp" like a madman, I found a few useful blog posts here and there about functional programming in Scala — what I'll call "Scala/FP" in this book — but those were too disconnected. One article covered Topic A, another covered Topic Z, and they were written by different authors with different experiences, so it was hard to find my way from A to Z. Besides being disjointed, they were often incomplete, or maybe they just assumed that I had some piece of knowledge that I didn't really have.

Another stumbling block is that experienced FP developers use generic types a *lot*. They also use the word "easy" when describing their code, as though saying "easy" is some sort of Jedi mind trick. For instance, this code — which I'll break down as you go through this book — was introduced with the text, "it's very easy to access and modify state":

```
def updateHealth(delta: Int): Game[Int] =
    StateT[IO, GameState, Int] { (s: GameState) =>

    val newHealth = s.player.health + delta
      IO((s.copy(player = s.player.copy(health = newHealth)), newHealth))

}
```

I don't know about you, but the first time I saw that code, the word *easy* is not what came to mind. What came to my mind were things like, "PHP is easy. Using setter methods to modify state is easy. Whatever that is … that's not easy."

Another problem with almost all of the Scala/FP resources is that they don't discuss functional input/output (I/O), or how to work with user interfaces. In this book I don't shy away from those topics: I write what I know about both of them.

Learning Haskell to learn FP

In the end, the only way I could learn FP was to buy four Haskell books, take a few weeks off from my regular work, and teach myself Haskell. Because Haskell is a "pure" FP language — and because most experienced Scala/FP developers spoke glowingly about Haskell — I assumed that by learning Haskell I could learn FP.

That turned out to be true. In Haskell the only way you can write code is by using FP concepts, so you can't bail out and take shortcuts when things get difficult. Because everything in Haskell is immutable, I was forced to learn about topics like recursion that I had avoided for most of my programming life. In the *beginning* this made things more difficult, but in the *end* I learned about the benefits of the new approaches I was forced to learn.

Once I understood Haskell, I went back to the Scala resources that I didn't like before and they suddenly made sense(!). But again, this only happened *after I took the time to learn Haskell*, a language I didn't plan on using in my work.

The purpose of this book

Therefore, my reasons for writing this book are:

- To save you the time of having to try to understand many different, unorganized, inconsistent Scala/FP blog posts
- To save you the time of "having to learn Haskell to learn FP," and then having to translate that Haskell knowledge back to Scala
- To try to make learning Scala/FP as simple as possible

> Don't get my statements about Haskell wrong: In the end, Haskell turned out to be a really interesting and even fun programming language. If I knew more about its libraries — or if it ran on the JVM and I could use the wealth of existing JVM libraries out there (most of which are not written in an FP style) — I'd be interested in trying to use it. That being said, I hope I can teach you what I learned about FP using only Scala.
>
> As a potential benefit of this book, if you already know Scala/OOP and are interested in learning Haskell, you can learn Scala/FP from this book, and then you'll find it much easier to understand Haskell.

2

Who This Book is For

> *"I never teach my pupils. I only attempt to provide the conditions in which they can learn."*
>
> Albert Einstein

I kept several audiences in mind as I wrote this book:

1. Developers who want a simple introduction to functional programming in Scala
2. Developers who are interested in writing "better" code
3. Parallel/concurrent application developers
4. "Big data" application developers
5. (Possibly) Upperclass college students

Here's a quick look at why I say that I wrote this book for these people.

1) Developers who want a simple introduction to FP

First, because this book started as a series of small notes I made for myself as I learned about functional programming in Scala, it's safe to say that I wrote it for someone like me who has worked with OOP in Java, but has only a limited FP background. Specifically, this is someone who became interested in Scala because of its clean, modern syntax, and now wants a "simple but thorough" introduction to functional programming in Scala.

Because I've also written programs in C, C++, Perl, Python, Ruby, and a few other programming languages, it's safe to say that this book is written with these programmers in mind as well.

2) Those interested in writing "better" code

At specific points in this book — such as (a) when writing about pure functions, (b) using `val` and not `var`, and (c) avoiding the use of `null` values — I also wrote this book for any developer that wants to write better code, where I define "better" as safer, easier to test, and more error-free. Even if you decide not to write 100% pure FP code, many FP techniques in this book demonstrate how you can make your functions and methods safer from bugs.

As a personal note, an ongoing theme in my programming life is that I want to be able to write applications faster, without sacrifing quality and maintainability. A selling point of FP is that it enables you to write safe functions — *pure* functions that rely only on their inputs to produce their outputs — that you can then combine together to create applications.

3) Parallel/concurrent developers

> Quiz: How many cores are in your smartphone? (This question is a tip of the cap to Derek Wyatt, who wrote about CPU cores and smartphones in his book, Akka Concurrency).

In addition to writing safer code, the "killer app" for FP since about 2005 is that CPUs aren't constantly doubling in speed any more. (See Herb Sutter's 2005 article, The Free Lunch is Over.) Because of this, CPU designers are adding more cores to CPUs to get more overall CPU cycles/second. Therefore, if you want your apps to run as fast as possible, you need to use *concurrent programming* techniques to put all of those cores to use, and the best way we know how to do that today is to use FP.

Two of my favorite ways of writing parallel/concurrent applications involve using Scala futures and the Akka messaging/actors framework. Not surprisingly, FP works extremely well with both of these approaches.

> Note that if quantum computers were available tomorrow, performance might no longer be an issue, but even in that world we'll still need to write concurrent applications, and as mentioned, FP is the best way to write parallel and concurrent applications today. I'll provide more support for that statement within this book, but one simple thing I can say now is that because there are no mutable variables in FP code, it's not possible to modify the same variable in different threads simultaneously.

4) "Big data" app developers

More recently, Dean Wampler gave a presentation titled, "Copious Data: The 'Killer App' for Functional Programming". My experience with Big Data applications is limited to processing large Apache access log records with Spark, but I can confirm that the code I wrote was a lot like algebra, where I passed data into pure functions and then used only the results from those functions. My code had no dependence on "side effects," such as using mutable variables or managing state.

5) Upperclass college students

As I wrote in the Scala Cookbook, because of its "power user" features, I don't think Scala is a good first language for a programmer to learn, and as a result of that, a book about Scala/FP is also not a good first programming book to read.

That being said, I hope this will be a good first FP book to read *after* a college student has experience with languages like C, Java, and Scala. When I started writing this book, my nephew was a senior in college and had some experience with C and Java, and as I reviewed the chapters I'd ask myself, "Would Tyler be able to understand this?"

Caution: Not for FP experts

Finally, as a result of the people I *have* written this book for, it should come as no surprise that this book is not written for FP experts and theorists. I offer no new theory in this book; I just try to explain functional programming using the Scala programming language in the simplest way I can.

3

Goals, Part 1: "Soft" Goals of This Book

> *"One learns by doing the thing."*
>
> Sophocles

Going through the thought process of "Why do I want to write a book about Scala/FP?" led me to develop my goals for this book. They are:

1. To introduce functional programming in Scala in a simple, thorough way, as though you and I are having a conversation.
2. To present the solutions in a systematic way. I want to introduce the material in the order in which I think you'll run into problems as you learn Scala/FP. In doing this, I break down complex code into smaller pieces so you can see how the larger solution is built from the smaller pieces.
3. To discuss the motivation and benefits of FP features. For me it wasn't always clear why certain things in FP are better, so I'll keep coming back to these two points.
4. Because it helps to see the big picture, I provide several small-but-complete Scala/FP example applications. Showing complete applications helps demonstrate how you can organize your FP applications, and work with issues like handling state, I/O, and user interfaces.
5. I hope to save you the time and effort of having to learn Haskell (or some other language) in order to learn FP.
6. I want to help you learn to "Think in FP." (More on this shortly.)

In general, I want to help you start writing FP code without having to learn a lot of mathematics, background theory, and technical jargon like that shown in Figure 3.1.

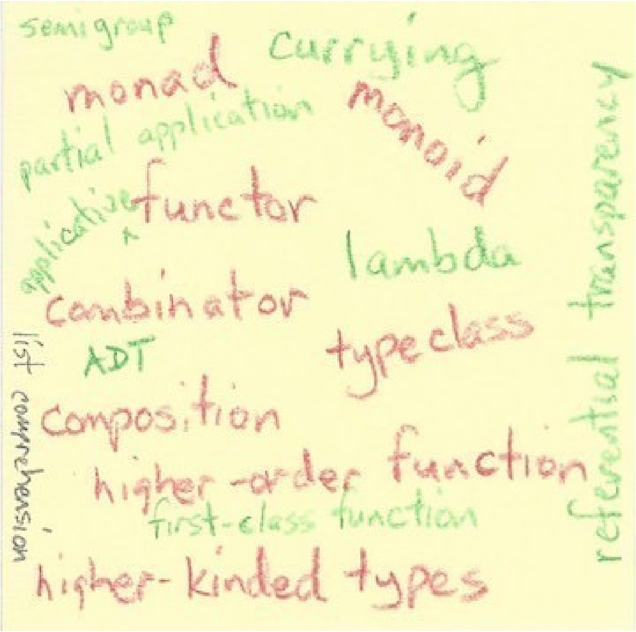


Figure 3.1: Examples of the "FP terminology barrier"

I refer to this as the "FP Terminology Barrier," and I'll discuss it more in an upcoming lesson.

While I generally avoid using technical jargon unless it's necessary, I do discuss many of these terms in the appendices, and I also provide references to resources where you can dive deeper into FP theory.

A word of caution: "The Learning Cliff"

When I took my first thermodynamics class in college, I learned the quote I shared at the beginning of this chapter:

> "One learns by doing the thing."

For me, this means that sometimes the only way you can learn something is to work on it with your hands. Until that first thermodynamics class I never really had to *do the thing* — work all of the exercises — to learn the material, but in that class I found out the hard way that there are times when I really have to dig in and "do the thing to learn the thing."

Another time when I had this same feeling was around the year 2009, when I started learning a content management system (CMS) named Drupal. During that time I came across the following image, where someone (whose name I can't find) made the point that Drupal didn't have a learning curve, but instead it had a *learning cliff*, which they depicted as shown in Figure 3.2.

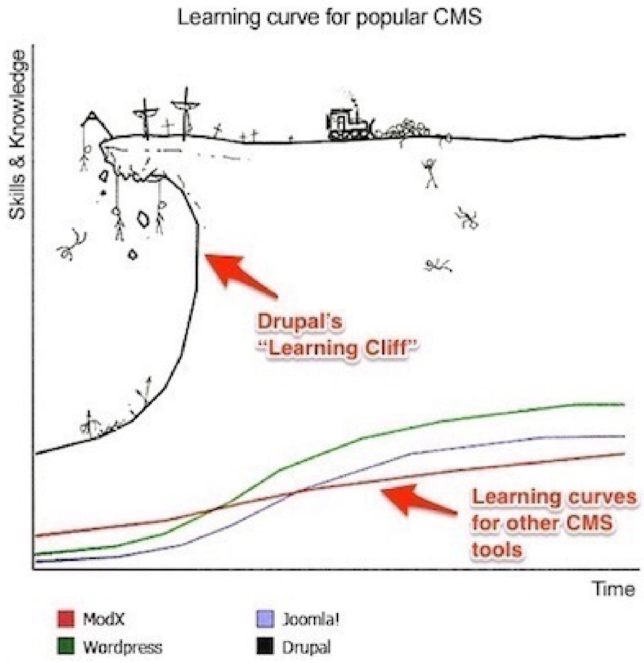


Figure 3.2: The Drupal "learning cliff" (original image source unknown).

There are two things to say about Drupal's learning cliff:

1. I found the diagram to be accurate.
2. Looking back on it, learning Drupal was one of the most beneficial things I've done for myself since that time. While getting started with Drupal was difficult, it ended up being a great investment of my time. The rewards were significant, and I have no regrets about spending the time it took to learn it. (If you're reading this text on a website, the page you're looking at was generated with Drupal.)

Just like FP, the problem with trying to learn Drupal is that when you get to a certain point (a) it feels like there are many things you need to learn simultaneously, and (b) it’s easy to take accidental detours on the learning path. As the image depicts, you either keep learning, or you fall off the cliff.

I hope that by breaking the Scala/FP learning process down into small, manageable pieces, this book can help you stay on the path, and make the Scala/FP learning curve more like other, normal learning curves.

Aside: Working hard to learn something new

If you’ve read the book, Einstein: His Life and Universe, by Walter Isaacson, you know that Albert Einstein had to essentially go back to school and learn a *lot* of math so he could turn the Theory of Special Relativity into the Theory of General Relativity.

He published the “Einstein field equations” (shown in Figure 3.3) in 1915, and other than trying to describe his theories to an advanced mathematician who could understand what he was talking about, there’s no way that Einstein could have developed these equations without buckling down and taking the time to learn the necessary math. (Even one of the smartest people in the history of Earth had to work hard to learn something new.)

$$\boxed{R_{\mu\nu} - \tfrac{1}{2}R\,g_{\mu\nu} + \Lambda g_{\mu\nu} = \frac{8\pi G}{c^4}T_{\mu\nu}}$$

Figure 3.3: The Einstein field equations.

More on Point #7: “Thinking in FP”

In this book I hope to change the way you think about programming problems. As at least one functional developer has said, when you “Think in FP,” you see an application as (a) data flowing into the application, (b) being transformed by a series of transformer functions, and (c) producing an output.

The first lessons in this book are aimed at helping you to "Think in FP" — to see your applications in this way, as a series of data flows and transformers.

As an example of this, when you get to the "FP is Like Writing Unix Pipelines" lesson, I'll share diagrams like Figure 3.4.

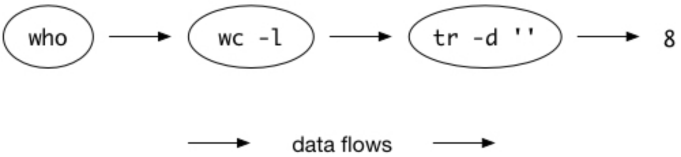


Figure 3.4: How data flows in a Unix pipeline.

As this image shows, a Unix pipeline starts with a source of data — the `who` command — and then transforms that data with one or more functions. Data flows through the pipeline in only one direction, from one function to the next, and data is never modified, it's only *transformed* from one format to another.

Another important part of "Thinking in FP" is that you'll find that FP function signatures are very important — much more important than they are in OOP code. I cover this in depth in the lesson, "Pure Function Signatures Tell All."

Summary

In summary, my goals for this book are:

1. To introduce functional programming in Scala in a simple, thorough way.
2. To present the solutions in a systematic way.
3. To discuss the motivation and benefits of FP features.
4. To share several small-but-complete Scala/FP applications to show you how they are organized.
5. To save you the time and effort of having to learn another programming language in order to understand Scala/FP.

6. In general, to help you “Think in FP.”

An important part of the learning process is having a “Question Everything” spirit, and I’ll cover that next.

4

Goals, Part 2: Concrete Goals

After I released Version 0.1.2 of this book, I realized that I should state my goals for it more clearly. I don't want you to buy or read a book that doesn't match what you're looking for. More accurately, I don't want you to be disappointed in the book because your expectations are different than what I deliver. Therefore, I want to state some very clear and measurable goals by which you can judge whether or not you want to buy this book.

A first concrete goal is this: If you have a hard time understanding the book, Functional Programming in Scala, I want to provide the background material that can help make that book easier to understand. That book is very good, but it's also a thin, densely-packed book, so if there are a few Scala features you don't know, it can be hard to keep up with it at times.

Second, the Introduction to Functional Game Programming talk at the 2014 LambdaConf was a big influence on me. I remember going to that talk and thinking, "Wow, I thought I knew Scala and a little bit about functional programming, but I have no idea what this guy is talking about." Therefore, a second concrete goal is to make all of that talk and its associated code understandable to someone who has zero to little background in functional programming. (That talk covers the `IO`, `State`, and `StateT` monads, and other FP features like *lenses*, so this is actually a pretty big goal.)

A third, slightly-less concrete goal is that if you have no background in FP, I want to make Scala/FP libraries like Cats and Scalaz more understandable. That is, if you were to look at those libraries without any sort of FP background, I suspect you'd be as lost as I was at that 2014 LambdaConf talk. But if you read this book, I think you'll understand enough Scala/FP concepts that you'll be able to further understand what those libraries are trying to achieve.

A fourth concrete goal is to provide you with all of the background knowledge you need — anonymous functions, type signatures, `for` expressions, classes that implement `map` and `flatMap`, etc. — so you can better understand the 128,000 *monad* tutorials that Google currently lists in their search results.

As one more point to further understand my goals, please read the "disclaimer" in the next chapter.

5

Goals, Part 3: A Disclaimer

As a bit of a warning, I want to be clear that this book is *very different* than the Scala Cookbook. The essence of the Cookbook is, “Here’s a common problem, and here’s a solution to that problem,” i.e., a series of recipes.

This book is completely different.

The “reporter” metaphor

I liken this book to being a reporter who goes to a foreign country that very few people seem to know about. Out of curiosity about what he has read and seen, the intrepid reporter goes to this foreign land to learn more about it. Nobody knows how the story is going to end, but the reporter promises to report the truth as he sees and understands it.

On his journey through this new land the reporter jots down many notes, especially as he has a few “Aha!” moments when he really grasps new concepts. Over time he tries to organize his notes so he can present them in a logical order, trying to translate what he has seen into English (and Scala) as simply and accurately as he can. In the end there’s no promise that the reporter is going to *like* what he sees, but he promises to report everything as clearly as he can.

A reporter is not a salesman

To be clear, there’s no promise of a happy ending in this story. The reporter isn’t trying to sell you on moving to this new land. (For all he knows, this new territory is full of Romulans or The Borg, and he may end up having to flee for his life.)

Instead of trying to *sell* you, the reporter aims to report what he sees as accurately as possible, hoping that — armed with this new knowledge — in the end you'll decide what's in your own best interests. Maybe you'll decide to move to this land, maybe you won't, but at least you'll be well-armed in making your decision.

A personal experience

As an example of how I think about this, many years ago I came close to moving to Santa Fe, New Mexico. As soon as I visited the town, I immediately fell in love with the plaza area, the food, and the architecture of the homes. But after thinking about the pros and cons more seriously, I decided not to move there. Instead, I decided to just vacation there from time to time, and also take home some nice souvenirs as I found them.

The same is true about this book: you may decide to move to this new land, or you may decide that you just like a few souvenirs. That choice is yours. My goal is to report what I find, as simply and accurately as I can.

6

Question Everything

> *"I have no special talent. I am only passionately curious."*
>
> Albert Einstein

A Golden Rule of this book is to always ask, "*Why?*" By this I mean that you should question everything I present. Ask yourself, "Why is this FP approach better than what I do in my OOP code?" To help develop this spirit, let's take a little look at what FP is, and then see what questions we might have about it.

What is FP?

I'll describe FP more completely in the "What is Functional Programming?" lesson, but for the moment let's use the following function of FP:

- FP applications consist of only immutable values and pure functions.
- *Pure function* means that (a) a function's output depends only on its input parameters, and (b) functions have no side effects, such as reading user input or writing output to a screen, disk, web service, etc.

While I'm intentionally keeping that definition short, it's a way that people commonly describe FP, essentially the FP elevator pitch.

What questions come to mind?

Given that description, what questions come to mind about FP?

Some of my first questions were:

- How can you possibly write an application without reading input or writing output?
- Regarding I/O:
  - How do I write database code?
  - How do I write RESTful code?
  - How do I write GUI code?
- If all variables are immutable, how do I handle changes in my code?
  - For instance, if I'm writing an order-entry system for a pizza store, what do I do when the customer wants to change their pizza crust or toppings in the middle of entering an order?

If you have a little more exposure to FP than I did, you might ask:

- Why is recursion better? Is it *really* better? Why can't I just use `var` fields inside my functions, as long as I don't share those `var`s outside the function scope?
- Is "Functional I/O" really better than "Traditional I/O"?

A little later you might ask:

- Are there certain applications where the FP approach is better? Or worse?

Decide for yourself what's better

Critical thinking is an important part of being a scientist or engineer, and I always encourage you to think that way:

> Is the approach I'm looking at better or worse than other options? If so, why?

When doing this I encourage you not to make any snap judgments. Just because you don't like something *initially* doesn't mean that thing is bad or wrong.

"The best idea wins"

With critical thinking you also need to tune out the people who yell the loudest. Just because they're loud, that doesn't mean they're right. Just focus on which ideas are the best.

In my book, A Survival Guide for New Consultants, I share this quote from famed physicist Richard Feynman:

> "The best idea wins."

He wrote this in one of his books, where he shared an experience of how Neils Bohr would seek out a very young Feynman during the building of the first atomic bomb. Bohr felt that the other scientists on the project were "Yes men" who would agree with anything he said, while Feynman was young, curious, and unintimidated. Because Feynman was only interested in learning and in trying to come up with the best solutions, he would tell Bohr exactly what he thought about each idea, and Bohr sought him out as a sounding board.

Feynman meant that you have to be able to have good, honest conversations with people about your ideas, and at the end of the day you have to put your ego aside, and the team should go forward with the best idea, no matter where it came from.

This goes back to my point: Don't blindly listen to people, especially the people who yell the loudest or those who can profit from selling you an idea. Put your critical thinking hat on, and make your own decisions.

A quick aside: Imperative programming

In the next sections I'll use the term "Imperative programming," so I first want to give you a definition of what it means.

With a few minor changes, Wikipedia offers this description: "*Imperative programming* is a programming paradigm that uses statements that change a program's state. It consists of a series of commands for the computer to perform. It focuses on describing the details of *how* a program operates."

This Quora page adds: “Imperative programming involves writing your program as a series of instructions (statements) that actively modify memory (variables, arrays). It focuses on ‘how,’ in the sense that you express the logic of your program based on how the computer would execute it.”

If you’ve ever disassembled a JVM *.class* file with `javap -c` to see code like this:

```
public void main(java.lang.String[]);
  Code:
     0: aload_0
     1: aload_1
     2: invokestatic  #60
     5: return
```

That’s the extreme of what they’re referring to: imperative programming at a very low level. This code tells the JVM *exactly* how it needs to solve the problem at hand.

A critical thinking exercise

To start putting your critical thinking skill to work, I’m going to show you two versions of the same algorithm. As you see the two algorithms, I want you to jot down any questions you have about the two.

First, here’s an imperative version of a `sum` method:

```
def sum(ints: List[Int]): Int = {
    var sum = 0
    for (i <- ints) {
      sum += i
    }
    sum
}
```

This code modifies a `var` field within a `for` loop — a very common pattern in imperative programming.

Next, here's a Scala/FP version of that same method:

```
def sum(xs: List[Int]): Int = xs match {
    case Nil => 0
    case x :: tail => x + sum(tail)
}
```

Notice that this method uses a `match` expression, has no `var` fields, and it makes a recursive call to `sum` in the last line of the method body.

Given those two versions of the same algorithm, what questions come to your mind?

My questions

The questions you have will depend heavily on your experience. If you're very new to Scala/FP your first question might be, "How does that second method even work?" (Don't worry, I'll explain it more in the lessons on writing recursive functions.)

I remember that some of my first questions were:

- What's wrong with the imperative approach? Who cares if I use a `var` field in a `for` loop inside a function? How does that affect anything else?
- Will the recursive function blow the stack with large lists?
- Is one approach faster or slower than the other?
- Thinking in the long term, is one approach more maintainable than the other?
- What if I want to write a "parallel" version of a sum algorithm (to take advantage of multiple cores); is one approach better than the other?

That's the sort of thinking I want you to have when you're reading this book: Question everything. If you think something is better, be honest, *why* do you think it's better? If you think it's worse, why is it worse?

In the pragmatic world I live in, if you can't convince yourself that a feature is better than what you already know, the solution is simple: Don't use it.

As I learned FP, some of it was so different from what I was used to, I found that *questioning everything* was the only way I could come to accept it.

"We write what we want, not how to do it"

As another example of having a questioning attitude, early in my FP learning process I read quotes from experienced FP developers like this:

> "In FP we don't tell the computer *how* to do things, we just tell it *what* we want."

When I read this my first thought was pretty close to, "What does that mean? You talk to the computer?"

I couldn't figure out what they meant, so I kept questioning that statement. Were they being serious, or was this just some sort of FP koan, trying to get you interested in the topic with a mysterious statement? It felt like they were trying to sell me something, but I was open to trying to understand their point.

After digging into the subject, I finally decided that the main thing they were referring to is that they don't write imperative code with `for` loops. That is, they don't write code like this:

```
def double(ints: List[Int]): List[Int] = {
    val buffer = new scala.collection.mutable.ListBuffer[Int]()
    for (i <- ints) {
        buffer += i * 2
    }
    buffer.toList
}

val newNumbers = double(oldNumbers)
```

Instead, they they write code like this:

```
val newNumbers = oldNumbers.map(_ * 2)
```

With a `for` loop you tell the compiler the exact steps you want it to follow to create the new list, but with FP you say, “I don’t care how `map` is implemented, I trust that it’s implemented well, and what I want is a new list with the doubled value of every element in the original list.”

In this example, questioning the “We write what we want” statement is a relatively minor point, but (a) I want to encourage a curious, questioning attitude, and (b) I know that you’ll eventually see that statement somewhere, and I wanted to explain what it means.

> In his book Programming Erlang, Joe Armstrong notes that when he was first taught object-oriented programming (OOP), he felt that there was something wrong with it, but because everyone else was “Going OOP,” he felt compelled to go with the crowd. Paraphrasing his words, if you’re going to work as a professional programmer and put your heart and soul into your work, make sure you believe in the tools you use.

What’s next?

In the next lesson I’m going to provide a few programming *rules* that I’ll follow in this book. While I’m generally not much of a “rules” person, I’ve found that in this case, having a few simple rules makes it easier to learning functional programming in Scala.

7

Rules for Programming in this Book

> *“Learn the rules like a pro, so you can break them like an artist.”*
>
> — Pablo Picasso

Alright, that’s enough of the “preface” material, let’s get on with the book!

As I wrote earlier, I want to spare you the route I took of, “You Have to Learn Haskell to Learn Scala/FP,” *but*, I need to say that I did learn a valuable lesson by taking that route:

> It’s extremely helpful to completely forget about several pieces of the Scala programming language as you learn FP in Scala.

Assuming that you come from an “imperative” and OOP background as I did, your attempts to learn Scala/FP will be hindered because *it is* possible to write both imperative code and FP code in Scala. Because you *can* write in both styles, what happens is that when things in FP start to get more difficult, it’s easy for an OOP developer to turn back to what they already know, rather than to try to navigate the “FP Learning Cliff.”

(I was a Boy Scout, if only briefly.)

To learn Scala/FP the best thing you can do is *forget* that the imperative options even exist. I promise you, Scout’s Honor, this will accelerate your Scala/FP learning process.

Therefore, to help accelerate your understanding of how to write FP code in Scala, this book uses only the following subset of the Scala programming language.

The rules

To accelerate your Scala/FP learning process, this book uses the following programming "rules":

1. There will be no `null` values in this book. We'll intentionally forget that there is even a `null` keyword in Scala.
2. Only *pure functions* will be used in this book. I'll define pure functions more thoroughly soon, but simply stated, (a) a pure function must always return the same output given the same input, and (b) calling the function must not have any side effects, including reading input, writing output, or modifying any sort of hidden state.
3. This book will only use immutable values (`val`) for all fields. There are no `var` fields in pure FP code, so I won't use any variables (`var`) in this book, unless I'm trying to explain a point.
4. Whenever you use an `if`, you must always also use an `else`. Functional programming uses only *expressions*, not *statements*.
5. We won't create "classes" that encapsulate data and behavior. Instead we'll create data structures and write pure functions that operate on those data structures.

The rules are for your benefit (really)

These rules are inspired by what I learned from working with Haskell. In Haskell the only way you can *possibly* write code is by writing pure functions and using immutable values, and when those *really are your only choices*, your brain quits fighting the system. Instead of going back to things you're already comfortable with, you think, "Hmm, somehow other people have solved this problem using only immutable values. How can I solve this problem using pure FP?" When your thinking gets to that point, your understanding of FP will rapidly progress.

If you're new to FP those rules may feel limiting — and you may be wondering how you can possibly get *anything* done — but if you follow these rules you'll find that they lead you to a different way of thinking about programming problems. *Because* of these rules your mind will naturally gravitate towards FP solutions to problems.

For instance, because you can't use a `var` field to initialize a mutable variable before a `for` loop, your mind will naturally think, "Hmm, what can I do here? Ah, yes, I can use recursion, or maybe a built-in collections method to solve this problem." By contrast, if you let yourself reach for that `var` field, you'll never come to this other way of thinking.

Not a rule, but a note: using ???

While I'm writing about what aspects of the Scala language I *won't* use in this book, it's also worth noting that I will often use the Scala `???` syntax when I first sketch a function's signature. For example, when I first start writing a function named `createWorldPeace`, I'll start to sketch the signature like this:

```
def createWorldPeace = ???
```

I mention this because if you haven't seen this syntax before you may wonder why I'm using it. The reason I use it is because it's perfectly legal Scala code; that line of code will compile just fine. Go ahead and paste that code into the REPL and you'll see that it compiles just like this:

```
scala> def createWorldPeace = ???
createWorldPeace: Nothing
```

However, while that code does *compile*, you'll see a long error message that begins like this if you try to *call* the `createWorldPeace` function:

```
scala.NotImplementedError: an implementation is missing
```

I wrote about the `???` syntax in a blog post titled, What does '???' mean in Scala?, but in short, Martin Odersky, creator of the Scala language, added it to Scala for teaching cases just like this. The `???` syntax just means, "The definition of this function is TBD."

> If you're interested in how language designers add features to a programming language, that blog post has a link to a really interesting discussion started by Mr. Odersky. He begins the thread by stating, "If people don't hold me back I'm going to add this (???) to Predef," and then the rest of the thread is an interesting back-and-forth discussion about the pros and cons of adding this feature to the Scala language, and possibly using other names for this feature, such as using TODO instead of ???.

Summary

In summary, the rules we'll follow in this book are:

1. There will be no `null` values.
2. Only *pure functions* will be used.
3. Immutable values will be used for all fields.
4. Whenever you use an `if`, you must always also use an `else`.
5. We won't create "classes" that encapsulate data and behavior.

What's next

Given these rules, let's jump into a formal definition of "functional programming."

8

One Rule for Reading this Book

In addition to the rules for *programming* in this book, there's one rule for *reading* this book:

> If you already understand the material in a lesson, move on to the next lesson.

Because I try to thoroughly cover everything you might possible need to know leading up to advanced topics like monads, there will probably be some lessons you don't need to read. For instance, you may already know that you can use functions as variables, how to write functions that have multiple parameter groups, etc.

Therefore, there's one simple rule for reading this book: If you already understand a topic — move on! (You can always come back and read it later if you feel like there's something you missed.)

9

What is “Functional Programming”?

> *“Object-oriented programming makes code understandable by encapsulating moving parts. Functional programming makes code understandable by minimizing moving parts.”*
>
> — Michael Feathers, author of Working Effectively with Legacy Code (via Twitter)

Defining “Functional Programming”

It’s surprisingly hard to find a consistent definition of *functional programming*. As just one example, some people say that functional programming (FP) is about writing *pure functions* — which is a good start — but then they add something else like, “The programming language must be lazy.” Really? Does a programming language really have to be lazy (non-strict) to be FP? (The correct answer is “no.”)

I share links to many definitions at the end of this lesson, but I think you can define FP with just two statements:

1. FP is about writing software applications using only pure functions.
2. When writing FP code you only use immutable values — `val` fields in Scala.

And when I say “only” in those sentences, I mean *only*.

You can combine those two statements into this simple definition:

> Functional programming is a way of writing software applications using only pure functions and immutable values.

Of course that definition includes the term "pure functions," which I haven't defined yet, so let me fix that.

A working definition of "pure function"

I provide a complete description of pure functions in the "Pure Functions" lesson, but for now, I just want to provide a simple working definition of the term.

A *pure function* can be defined like this:

- The output of a pure function depends only on (a) its input parameters and (b) its internal algorithm.
    - This is unlike an OOP method, which can depend on other fields in the same class as the method.
- A pure function has no side effects, meaning that it does not read anything from the outside world or write anything to the outside world.
    - It does not read from a file, web service, UI, or database, and does not write anything either.
- As a result of those first two statements, if a pure function is called with an input parameter `x` an infinite number of times, it will always return the same result `y`.
    - For instance, any time a "string length" function is called with the string "Alvin", the result will always be `5`.

As a few examples, Java and Scala functions like these are pure functions:

- `String` uppercase and lowercase methods
- `List` methods like `max`, `min`

- `Math.sin(a)`, `Math.cos(a)`

In fact, because the Java `String` class and Scala `List` class are both immutable, all of their methods act just like pure functions.

Even complex algorithms like checksums, encodings, and encryption algorithms follow these principles: given the same inputs an infinite number of times, they always return the same result.

Conversely, functions like these are *not* pure functions:

- `System.currentTimeMillis`
- `Random` class methods like `next`, `nextInt`
- I/O methods in classes like `File` and `HttpURLConnection` that read and write data

The first two examples yield different results almost every time they are called, and I/O functions are impure because they have *side effects* — they communicate with the outside world to send and receive data.

Note 1: Higher-Order Functions are a great FP language feature

If you're not familiar with the term Higher-Order Function (HOF), it basically means that (a) you can treat a function as a value (`val`) — just like you can treat a `String` as a value — and (b) you can pass that value into other functions.

In writing good FP code, you pass one function to another so often that I'm tempted to add HOFs as a requirement to my definition. But in the end, you can write FP code in languages that don't support HOFs, including Java. Of course that will be painful and probably very verbose, but you can do it.

Therefore, I don't include HOFs in my definition of functional programming. In the end, HOFs are a *terrific* FP language feature, and they make Scala a *much* better FP language than Java, but it's still just a *language feature*, not a part of the core definition of functional programming.

> Note: I provide a more complete HOF definition in the glossary at the end of this book.

Note 2: Recursion is a by-product

Sometimes you'll see a definition of FP that states, "Recursion is a requirement of functional programming." While it's true that pure FP languages use recursion, the need for recursion is a *by-product* of my FP definition.

Once you dig into FP, you'll see that if you only use pure functions and immutable values, the *only* way you can do things like "calculate the sum of a list" is by using recursion. Therefore, it's a by-product of my definition, not a part of the definition.

> (I discuss this more in the recursion lessons.)

Proof: Wikipedia's FP definition

When you google "functional programming definition," the first link that currently shows up is from Wikipedia, and their definition of FP backs up my statements. The first line of their definition begins like this:

> "In computer science, functional programming is a programming paradigm — a style of building the structure and elements of computer programs — that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data."

So, yes, FP is made of (a) pure functions and (b) immutable data. (Their "mathematical functions" are equivalent to my pure functions.)

As proof for another assertion I made earlier, that Wikipedia page also elaborates on features that make an FP language easier to use — such as being able to treat functions as values — where they state, "Programming in a functional style can also be accomplished in languages that are not specifically designed for functional programming." (Think Java.)

Proof: A wonderful quote from Mary Rose Cook

When I first started learning FP, I was aware that pure functions were important, but this point was really driven home when I came across an article titled A Practical Introduction to Functional Programming by Mary Rose Cook.

Ms. Cook used to work at the Recurse Center (formerly known as "Hacker School") and now works at Makers Academy, and in her "Practical Introduction to FP" essay, she refers to using only pure functions as a *Guide Rope* to learning FP:

> "When people talk about functional programming, they mention a dizzying number of 'functional' characteristics. They mention immutable data, first class functions, and tail call optimisation. These are *language features* that aid functional programming."

> "They mention mapping, reducing, pipelining, recursing, currying and the use of higher order functions. These are *programming techniques* used to write functional code."

> "They mention parallelization, lazy evaluation, and determinism. These are advantageous properties of functional programs."

> "Ignore all that. Functional code is characterised by one thing: *the absence of side effects*. It (a pure function) doesn't rely on data outside the current function, and it doesn't change data that exists outside the current function. Every other 'functional' thing can be derived from this property. Use it as a guide rope as you learn."

When she writes about the "absence of side effects," she's referring to building applications from pure functions.

Her guide rope statement is so good, it bears repeating:

> "Functional code is characterised by one thing: the absence of side effects."

When I first read this quote, the little light bulb went on over my head and I began focusing even more on writing *only* pure functions.

If you think about it, this statement means exactly what I wrote at the beginning of this lesson:

> Functional programming is a way of writing software applications using only pure functions and immutable values.

That's great … but why immutable values?

At this point you might be saying, "Okay, I buy the 'pure functions' portion of your definition, but what does *immutable values* have to do with this? Why can't my variables be mutable, i.e., why can't I use `var`?"

The best FP code is like algebra

I dig into this question in the "FP is Like Algebra" lesson, but the short answer here is this:

> The best FP code is like algebra, and in algebra you never re-use variables. And not re-using variables has many benefits.

For example, in Scala/FP you write code that looks like this:

```
val a = f(x)
val b = g(a)
val c = h(b)
```

When you write simple expressions like this, both you and the compiler are free to rearrange the code. For instance, because `a` will always be *exactly* the same as `f(x)`, you can replace `a` with `f(x)` at any point in your code.

The opposite of this is also true: `a` can always be replaced with `f(x)`. Therefore, this equation:

```
val b = g(a)
```

is exactly the same as this equation:

```
val b = g(f(x))
```

Continuing along this line of thinking, because `b` is *exactly* equivalent to `g(f(x))`, you can also state `c` differently. This equation:

```
val c = h(b)
```

is exactly the same as this equation:

```
val c = h(g(f(x)))
```

From a programming perspective, knowing that you can *always* replace the immutable values `a` and `b` with their equivalent functions (and vice-versa) is extremely important. If `a` and `b` had been defined as `var` fields, I couldn't make the substitutions that I did. That's because with mutable variables you can't be certain that later in your program `a` is still `f(x)`, and `b` is still `g(a)`. However, because the fields *are* immutable, you can make these algebraic substitutions.

FP code is easier to reason about

Furthermore, because `a` and `b` can never change, the code is easier to reason about.

With `var` fields you always have to have a background thread running in your brain, "Is `a` reassigned somewhere else? Keep an eye out for it."

But with FP code you never have to think, "I wonder if `a` was reassigned anywhere?" That thought never comes to mind. `a` is the same as `f(x)`, and that's all there is to it, end of story. They are completely interchangeable, just like the algebra you knew in high school.

> To put this another way, in algebra you never reassign variables, so it's obvious that the third line here is a mistake:
>
> ```
> a = f(x)
> b = g(a)
> a = h(y)      # d'oh -- `a` is reassigned!
> c = i(a, b)
> ```
>
> Clearly no mathematician would ever do that, and because FP code is like algebra, no FP developer would ever do that either.

Another good reason to use immutable values

Another good reason to use only immutable values is that mutable variables (`var` fields) don't work well with parallel/concurrent applications. Because concurrency is becoming more important as CPUs use more cores, I discuss this in the "Benefits of Functional Programming" and "Concurrency" lessons.

> As a prelude to those lessons, in the article, The Downfall of Imperative Programming, Bartosz Milewski writes, "Did you notice that in the definition of 'data race' there's always talk of *mutation*?"

> As programmers gain more experience with FP, their code tends to look more like this expression:
>
> ```
> val c = h(g(f(x)))
> ```

> While that's cool — and it's also something that your brain becomes more comfortable with over time — it's also a style that makes it harder for new FP developers to understand. Therefore, in this book I write most code in the simple style first:
>
> ```
> val a = f(x)
> val b = g(a)
> val c = h(b)
> ```
>
> and then conclude with the reduced code at the end:
>
> ```
> val c = h(g(f(x)))
> ```
>
> As that shows, when functions are pure and variables are immutable, the code is like algebra. This is the sort of thing we did in high school, and it was all very logical. (FP developers refer to this sort of thing as "evaluation" and "substitution.")

Summary

In this lesson, I defined functional programming like this:

> Functional programming is a way of writing software applications using only pure functions and immutable values.

To support that, I also defined pure function like this:

- The output of a pure function depends only on (a) its input parameters and (b) its internal algorithm.
- A pure function has no side effects, meaning that it does not read anything from the outside world or write anything to the outside world.
- As a result of those first two statements, if a pure function is called with an input parameter `x` an infinite number of times, it will always return the same result `y`.

I noted that higher-order functions (HOFs) are a terrific FP language feature, and also stated that recursion is a by-product of the definition of FP.

I also briefly discussed some of the benefits of immutable values (and FP in general):

- The best FP code is like algebra
- Pure functions and immutable values are easier to reason about
- Without much support (yet), I stated that immutable values make parallel/concurrent programming easier

See also

- A Postfunctional Language, a scala-lang.org post by Martin Odersky
- The docs.scala-lang.org definition of functional style
- The Wikipedia definition of FP
- The Clojure definition of FP
- The Haskell definition of FP
- The "Creative Clojure" website agrees with my definition of functional programming
- Information about FP in the Real World Haskell book
- Here's the msdn.microsoft.com definition of FP
- Functional programming on c2.com
- A practical introduction to functional programming
- An intro to FP on the "Learn You a Haskell for Great Good" website
- Stack Exchange thread
- Why do immutable objects enable functional programming?
- The "Benefits of Functional Programming" lesson in this book
- The "Concurrency" lesson in this book

10

What is This Lambda You Speak Of?

> *“It takes a wise man to learn from his mistakes, but an even wiser man to learn from others.”*
>
> — Zen Proverb

Goals

Once you get into FP, you’ll quickly start hearing the terms “lambda” and “lambda calculus.” The goal of this chapter is to provide background information on where those names come from, and what they mean.

This chapter is mostly about the history of functional programming, so for people who don’t like history, I first share a short lesson that just explains those terms. After that, I add a full version that discusses the invention of the lambda calculus, several key people in the FP history, and languages like Lisp, Haskell, and Scala.

The short story

For those who don’t like history, this is the shortest possible “history of functional programming” I can provide that explains where the terms lambda and lambda calculus come from.

"Lambda"

Back in the 1930s, Alonzo Church was studying mathematics at Princeton University and began using the Greek symbol λ — "lambda" — to describe ideas he had about these things called *functions*. Because his work preceded the development of the first electronic, general-purpose computer by at least seven years, you can imagine him writing that symbol on chalkboards to describe his concept of functions.

So, historically speaking, that's the short story of where the term "lambda" comes from; it's just a symbol that Mr. Church chose when he first defined the concept of a function.

Fast-forward to today, and these days the name lambda is generally used to refer to anonymous functions. That's all it means, and it bears highlighting:

> In modern functional programming, lambda means "anonymous function."

If you're familiar with other programming languages, you may know that Python and Ruby use the keyword `lambda` to define anonymous functions.

> If you're not familiar with anonymous functions, I wrote about them in the Scala Cookbook, and I also provide an overview of them in the appendices of this book.

The term "lambda calculus"

As an aerospace engineer, I always thought the name "calculus" referred to the form of mathematics that has to do with infinitesimal changes and derivatives, but the name calculus also has a broader meaning. The word calculus can mean "a formal system," and indeed, that's how Wikipedia defines lambda calculus:

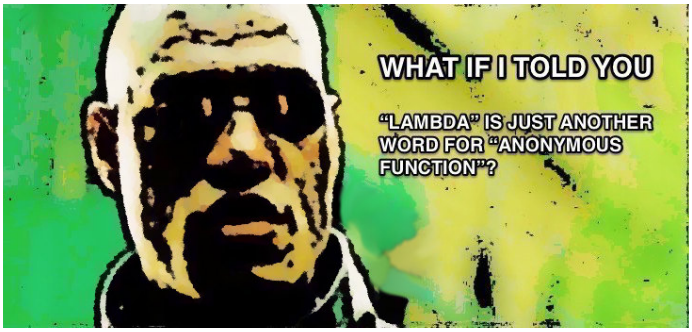


Figure 10.1: "Lambda" is just another name for "anonymous function"

> "Lambda calculus (also written as λ-calculus) is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution."

So we have:

- lambda means "anonymous function," and
- calculus means "a formal system"

Therefore, the term *lambda calculus* refers to "a formal way to think about functions."

That same Wikipedia link states this:

> "Lambda calculus provides a theoretical framework for describing functions and their evaluation. Although it is a mathematical abstraction rather than a programming language, it forms the basis of almost all functional programming languages today."

When I first started learning about functional programming, I found these terms to be a little intimidating, but as with most FP terms, they're just uncommon words for talking about "functions and their evaluation."

If you're interested in the deeper history of FP, including a guy named Haskell Curry, the relationship between FORTRAN and FP, and languages like Lisp, Haskell, Scala, and Martin Odersky's work that led to the creation of Scala, continue reading the next section. Otherwise feel free to move on to the next chapter.

The Longer Story (History)

Back in the 1930s — 80+ years ago — gasoline cost 17 cents a gallon, World War II hadn't started yet (not until 1939, officially), the United States was in the midst of the Great Depression (1929-1939), and a man by the name of Alonzo Church was studying mathematics at Princeton University along with other legendary figures like Alan Turing (who finished his PhD under Church) and John von Neumann.

Mr. Church spent two years as a National Research Fellow and a year at Harvard, and was interested in things like mathematical and symbolic logic. In 1956 wrote a classic book titled, "Introduction to Mathematical Logic."

In 1936 Mr. Church released his work on "lambda calculus," and it turned out to be a very important work indeed. Think about it: How many other papers from 1936 do you know that influence life today? His biography page at www-history.mcs.st-andrews.ac.uk states:

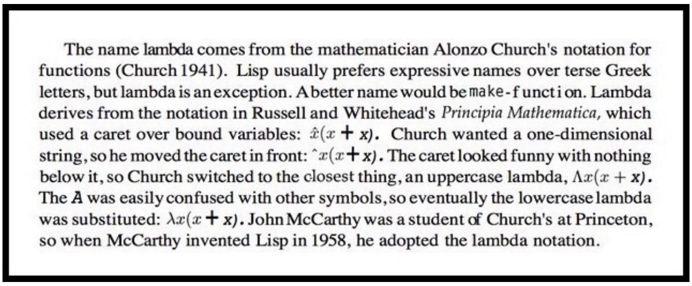
> "Church's great discovery was lambda calculus … his remaining contributions were mainly inspired afterthoughts in the sense that most of his contributions, as well as some of his pupils', derive from that initial achievement."

Wikipedia previously stated that the name "lambda" was arbitrary:

> "The name derives from the Greek letter lambda (λ) used to denote binding a variable in a function. The letter itself is arbitrary and has no special meaning."

However, other research shows that the choice of the name wasn't entirely arbitrary. After all, this is the man who founded the "Journal of Symbolic Logic," so I personally doubted it was completely arbitrary. I imagine him drawing this symbol on chalkboards and papers in the 1930s, so my first guess was that he wanted a symbol that was easy to read and write, but fortunately you don't have to rely on my guesswork.

The book “Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp,” by Peter Norvig discusses the origin of the λ symbol, as shown in Figure 10.2.



The name lambda comes from the mathematician Alonzo Church's notation for functions (Church 1941). Lisp usually prefers expressive names over terse Greek letters, but lambda is an exception. A better name would be `make-function`. Lambda derives from the notation in Russell and Whitehead's *Principia Mathematica,* which used a caret over bound variables: $\hat{x}(x + x)$. Church wanted a one-dimensional string, so he moved the caret in front: $\hat{}x(x + x)$. The caret looked funny with nothing below it, so Church switched to the closest thing, an uppercase lambda, $\Lambda x(x + x)$. The Λ was easily confused with other symbols, so eventually the lowercase lambda was substituted: $\lambda x(x + x)$. John McCarthy was a student of Church's at Princeton, so when McCarthy invented Lisp in 1958, he adopted the lambda notation.

Figure 10.2: The origin of the λ symbol (by Peter Norvig)

Note that Mr. Norvig also states that a better name for lambda would be *make function*.

As mentioned, Mr. Church introduced the world to the “lambda calculus” in 1936. On his biography page, Wikipedia describes his work like this:

> “The lambda calculus emerged in his 1936 paper showing the unsolvability of the Entscheidungsproblem. This result preceded Alan Turing’s work on the halting problem, which also demonstrated the existence of a problem unsolvable by mechanical means. Church and Turing then showed that the lambda calculus and the Turing machine used in Turing’s halting problem were equivalent in capabilities, and subsequently demonstrated a variety of alternative ‘mechanical processes for computation.’ This resulted in the Church–Turing thesis.”

The Wikipedia functional programming page also states:

“Functional programming has its roots in lambda calculus, a formal system developed in the 1930s to investigate computability, the Entscheidungsproblem, function definition, function application, and recursion. *Many functional programming languages can be viewed as elaborations on the lambda calculus.*”

The book Becoming Functional states that lambda calculus introduced the concept of passing a function to a function. I cover this topic in the Scala Cookbook, and discuss it in several lessons in this book.

The 1950s and Lisp

While Mr. Church's lambda calculus was well known in its time, it's important to note that the ENIAC — generally recognized as the world's first electronic, general-purpose computer — wasn't put into action until 1946, and it fit Alan Turing's ideas better than Mr. Church's.

> The name Lisp is derived from "LISt Processor."

But then in the 1958, MIT professor John McCarthy, a former student of Mr. Church, introduced a computer programming language named Lisp, which was "an implementation of Mr. Church's lambda calculus that worked on von Neumann computers."

That second Wikipedia link describes the importance of the Lisp programming language:

> "Lisp was originally created as a practical mathematical notation for computer programs, influenced by the notation of Alonzo Church's lambda calculus. It quickly became the favored programming language for artificial intelligence (AI) research. As one of the earliest programming languages, Lisp pioneered many ideas in computer science, including tree data structures, automatic storage management, dynamic typing, conditionals, higher-order functions, recursion, and the self-hosting compiler."

That's pretty impressive for a programming language created in the 1950s. (Beatlemania didn't start until 1963, few people knew the Rolling Stones before 1965, and color television wouldn't become popular in the United States until the mid-1960s.)

As a last note about Lisp, famed programmer Eric Raymond, author of The Cathedral and the Bazaar, wrote an article titled How to become a hacker, where he wrote this about Lisp:

> "LISP is worth learning for a different reason — the profound enlightenment experience you will have when you finally get it. That experience will make you a better programmer for the rest of your days, even if you never actually use LISP itself a lot."

> One of the most interesting computer programming books I've ever come across is The Little Schemer. (Scheme is a dialect of Lisp.) The Little Schemer is written in a conversational style between a student and a teacher, where the teacher's main goal is to get the student to think recursively and see patterns. Another book named Land of Lisp may hold the distinction as being the programming book that looks most like a cartoon. It's another good resource, and it's a much more complete introduction to the language than The Little Schemer.
>
> If you happen to work with Gimp (GNU Image Manipulation Program) and want to automate your tasks, you'll find that it supports a scripting language named Script-Fu by default. Script-Fu is a dialect of Scheme.

John Backus, FORTRAN, and FP

Figure 10.3: John Backus

In 1977, John Backus won a Turing Award for his lecture, "Can Programming Be

Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs." With minor edits, that link states:

> "Backus later worked on a 'function-level' programming language known as FP … sometimes viewed as Backus's apology for creating FORTRAN, this paper did less to garner interest in the FP language than to spark research into functional programming in general."

The Wikipedia functional programming page adds:

> "He defines functional programs as being built up in a hierarchical way by means of 'combining forms' that allow an 'algebra of programs'; in modern language, this means that functional programs follow the principle of compositionality."

(Note: I write much more about "composition" in the lessons of this book.)

> I created the sketch of John Backus from his image on ibm.com.

Mr. Backus did much more than this, including his work on the ALGOL 60 programming language, and creation of the Backus-Naur Form (BNF). But in terms of functional programming, his 1977 lecture was an impetus for additional research.

Erlang

Way back in 1986, I was mostly interested in playing baseball, and programmers for a company named Ericsson created a programming language named Erlang, which was influenced by Prolog. Wikipedia states, "In 1998 Ericsson announced the AXD301 switch, containing over a million lines of Erlang and reported to achieve a high availability of nine '9's."

Erlang is not a pure functional programming language like Haskell, but it's an actor-based, message-passing language. If you've heard that term before, that may be because the Akka actor library was inspired by Erlang. If you've used Akka, you know that one actor can't modify the state of another actor, and in this regard, the Erlang message-passing architecture uses FP concepts.

Joe Armstrong is a famous Erlang programmer (and co-creator of the language), and in his book, Programming Erlang, he writes:

"In Erlang it's OK to mutate state within an individual process but not for one process to tinker with the state of another process … processes interact by one method, and one method only, by exchanging messages. *Processes share no data with other processes*. This is the reason why we can easily distribute Erlang programs over multicores or networks."

In that statement, Mr. Armstrong's "processes" are equivalent to Akka actors, so the same statement can be made: "Actors share no data with other actors, and because of this we can easily distribute Akka programs over multicores or networks." As you'll see in the lessons to come, using only immutable values lets us say the same things about pure FP applications.

Haskell

Haskell Brooks Curry (1900-1982) has the distinction of having three programming languages named after him (Haskell, Brook, and Curry). In addition to those, the process of "currying" is also named after him.

Wikipedia states:

> "The focus of Curry's work were attempts to show that combinatory logic could provide a foundation for mathematics … By working in the area of Combinatory Logic for his entire career, Curry essentially became the founder and biggest name in the field … In 1947 Curry also described one of the first high-level programming languages."

Mathematics Teacher Enjoys Bird Watching

University Park—Looking for a hobby? One that might give you the chance to rub elbows with royalty, get you out in the fresh air, and provide some cliff-hanging adventures, all at little or no expense? Then try bird watching.

That's the suggestion of Dr. Haskell B. Curry, Evan Pugh research professor of mathematics at the Pennsylvania State University.

Dr. Curry, founder of the State College Bird Watchers Club, has been observing the antics of the feathered creatures for more than 50 years now and is probably one of the longest-term bird watchers in the state.

And he has a stack of records nearly three feet tall to prove it, some dating back to 1916, reporting the sighting of nearly 150,000 separate birds.

But the true adventure story of decades of trampling through woods in search of birds isn't shown in the records.

Recalls Oslo Visit

For instance, Dr. Curry remembers the time he visited Oslo, Norway, in 1928 and decided to do some bird watching. In pursuit of his quarry, Dr. Curry jumped over a high fence onto a sprawling estate.

"No sooner had I crossed the fence than a gardener came running toward me with two enormous dogs," Dr. Curry

DR. HASKELL B. CURRY

The fact that Dr. Curry has had minor mishaps on several occasions hasn't dampened his enthusiasm for bird watching one bit.

Figure 10.4: Haskell Curry

(The newspaper image of Haskell Curry comes from this catonmat.net page.)

For a brief history of the Haskell programming language, I'll turn things over to the book, Learn You a Haskell for Great Good!:

> "Haskell was made by some really smart guys (with PhDs). Work on Haskell began in 1987 when a committee of researchers got together to design a kick-ass language. In 2003 the Haskell Report was published, which defines a stable version of the language."

For a longer and more technical account of Haskell's history, I recommend searching for a PDF titled, "A History of Haskell: Being Lazy With Class," by Paul Hudak, John Hughes, and Simon Peyton Jones, three of the co-creators of Haskell. In that paper you'll learn a few more things, including that Haskell was inspired by a programming language named Miranda.

In that paper you'll also find this quote from Virginia Curry, Haskell Curry's wife:

> "You know, Haskell actually never liked the name *Haskell*."

If you want to learn Haskell I highly recommend starting with that book. Real World Haskell is another good resource.

The maturation of the Haskell language and the nearly simultaneous introduction of multicore CPUs in mainstream computing devices has been a significant driving force for the increasing popularity of FP. Because CPUs no longer double in speed every two years, they now include multiple cores and Hyper-threading technology to provide performance gains. This makes multicore (concurrent) programming important, and as luck would have it, pure functions and immutable values make concurrent programming easier.

> As Joe Armstrong writes in his 2013 book Programming Erlang, "Modern processors are so fast that a single core can run four hyperthreads, so a 32-core CPU might give us an equivalent of 128 threads to play with. This means that 'a hundred times faster' is within striking distance. A factor of 100 does get me excited. All we have to do is write the code."

Martin Odersky and Scala

Martin Odersky was born in 1958, and received his PH.D. under the supervision of Niklaus Wirth (who is best known as a programming language designer, including the Pascal language, and received the Turing Award in 1984).

Mr. Odersky is generally best known for creating Scala, but before that he also created a language named Pizza, then Generic Java, and the `javac` compiler. With a little bit of editing for conciseness, the "Scala prehistory" page on scala-lang.org states:

> "In 1999, after he joined EPFL, the direction of his work changed a bit. The goal was still to combine functional and object-oriented programming, but without the restrictions imposed by Java. The first step was Funnel, a minimalist research language based on functional nets … Funnel was pleasingly pure from a language design standpoint, with very

Figure 10.5: Martin Odersky

> few primitive language features. Almost everything, including classes and pattern matching, would be done by libraries and encodings.”
>
> “However, it turned out that the language was not very pleasant to use in practice. Minimalism is great for language designers but not for users … The second — and current — step is Scala, which took some of the ideas of Funnel and put them into a more pragmatic language with special focus on interoperability with standard platforms. Scala’s design was started in 2001. A first public release was done in 2003. In 2006, a second, redesigned version was released as Scala v 2.0.”

> I created the sketch of Martin Odersky from the image on his Wikipedia page.

Today

Skipping over a few important programming languages like ML and OCaml and fast-forwarding to the here and now, in 2016 there are quite a few pure and impure functional programming languages, including Haskell, Erlang, Lisp/Scheme variants, F# ("a .NET implementation of OCaml"), Clojure (a dialect of Lisp), and of course, Scala. (My apologies to any languages I omitted.)

> On their "programming languages by type" page, Wikipedia provides a list of functional programming languages that are considered "pure" and "impure."

Figure 10.6 shows a rough timeline of some of the important events in the history of functional programming.

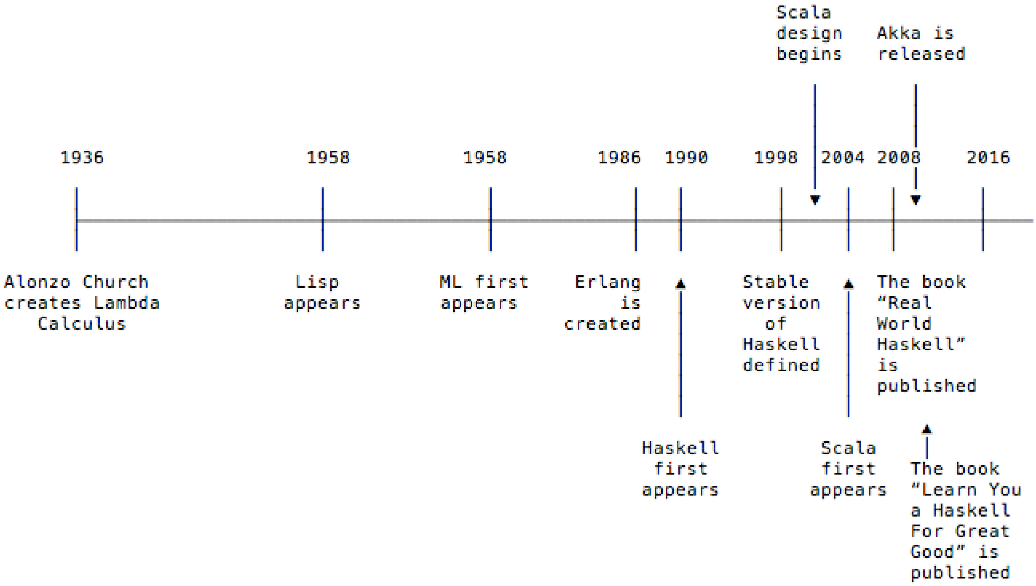


Figure 10.6: Timeline of events in FP history

One last point

It's important to note that Mr. Church most likely had no interest in things like (a) maintaining state over long periods of time, (b) interacting with files (reading and writing), and (c) networking. In fact, I'm pretty sure that the concept of a "file" had not been invented in the 1930s; packet-switching networks weren't invented until the late 1960s; and DARPA didn't adopt TCP/IP until 1983.

I mention this because while lambda calculus is important as “a theoretical framework for describing functions and their evaluation,” Mr. Church never said, “Let me tell you exactly how to work with files, GUIs, databases, web services, and maintaining state in functional applications … (followed by his solutions).”

This is important, because as mentioned, *pure functional programs* consist of only immutable values and pure functions. By definition, they can’t have I/O.

As an example of this problem (I/O in an FP language), the C programming language — created between 1969 and 1973 — could handle I/O, but here’s what the Wikipedia monad page states about Haskell, I/O, and monads:

> “Eugenio Moggi first described the general use of monads to structure programs in 1991. Several people built on his work … early versions of Haskell used a problematic ‘lazy list’ model for I/O, and Haskell 1.3 introduced monads as a more flexible way to combine I/O with lazy evaluation.”

When I write about monads later in this book, I like to remember that lambda calculus was invented in 1936, but monads weren’t described (invented) until 1991, and weren’t added to Haskell until version 1.3, which was released in 1998. That’s 62 years in between (a) lambda calculus and (b) monads to handle I/O in Haskell.

If you like history …

If you like history, Walter Isaacson’s book, The Innovators: How a Group of Hackers, Geniuses, and Geeks Created the Digital Revolution is a detailed history of the computer technology, tracing it all the way back to Ada Lovelace in the 1840s. The audiobook version of The Innovators is over 17 hours long, and I listened to it while driving across the United States in 2015, and I highly recommend it. (His biographies of Albert Einstein and Steve Jobs are also excellent.)

See also

- As I wrote the first draft of this chapter, Jamie Allen, Senior Director of Global Services for Lightbend, tweeted "When I need to break a problem into functions, thinking in LiSP helps me tremendously."
- My first exposure to the history of functional programming came in an article titled, Functional Programming for the Rest of Us
- Alonzo Church
- A biography of Alonzo Church
- Functional Programming (Wikipedia)
- ENIAC, the first computer in the world
- Haskell Curry
- Lambda mean anonymous function (Stack Overflow)
- Lambda mean anonymous function (Stack Exchange)
- Lambda in Python
- Lambda in Ruby (rubymonk)
- Lambda the Ultimate (website)
- "Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp," by Peter Norvig
- The history of Erlang
- "In many ways, F# is essentially a .Net implementation of OCaml"

11

The Benefits of Functional Programming

> *"Functional programming is often regarded as the best-kept secret of scientific modelers, mathematicians, artificial intelligence researchers, financial institutions, graphic designers, CPU designers, compiler programmers, and telecommunications engineers."*
>
> The Wikipedia F# page

As I write about the benefits of functional programming in this chapter, I need to separate my answers into two parts. First, there are the benefits of *functional programming in general*. Second, there are more specific benefits that come from using *functional programming in Scala*. I'll look at both of these in these chapter.

Benefits of functional programming in general

Experienced functional programmers make the following claims about functional programming, regardless of the language they use:

1. Pure functions are easier to reason about
2. Testing is easier, and pure functions lend themselves well to techniques like property-based testing
3. Debugging is easier
4. Programs are more bulletproof
5. Programs are written at a higher level, and are therefore easier to comprehend
6. Function signatures are more meaningful
7. Parallel/concurrent programming is easier

I'll discuss these benefits in this chapter, and then offer further proof of them as you go through this book.

Benefits of functional programming in Scala

On top of those benefits of functional programming in general, Scala/FP offers these additional benefits:

8. Being able to (a) treat functions as values and (b) use anonymous functions makes code more concise, and still readable
9. Scala syntax generally makes function signatures easy to read
10. The Scala collections' classes have a very functional API
11. Scala runs on the JVM, so you can still use the wealth of JVM-based libraries and tools with your Scala/FP applications

In the rest of this chapter I'll explore each of these benefits.

1) Pure functions are easier to reason about

The book, Real World Haskell, states, "Purity makes the job of understanding code easier." I've found this to be true for a variety of reasons.

First, pure functions are easier to reason about because you know that they can't do certain things, such as talk to the outside world, have hidden inputs, or modify hidden state. Because of this, you're guaranteed that their function signatures tell you (a) exactly what's going into each function, and (b) coming out of each function.

In his book, Clean Code, Robert Martin writes:

> "The ratio of time spent reading (code) versus writing is well over 10 to 1 … (therefore) making it easy to read makes it easier to write."

I suspect that this ratio is lower with FP. Because pure functions are easier to reason about:

- I spend less time "reading" them.
- I can keep fewer details in my brain for every function that I read.

This is what functional programmers refer to as "a higher level of abstraction."

Because I can read pure functions faster and use less brain memory per function, I can keep more overall logic in my brain at one time.

Several other resources support these statements. The book Masterminds of Programming includes this quote: "As David Balaban (from Amgen, Inc.) puts it, 'FP shortens the brain-to-code gap, and that is more important than anything else.' "

In the book, Practical Common Lisp, Peter Seibel writes:

> "Consequently, a Common Lisp program tends to provide a much clearer mapping between your ideas about how the program works and the code you actually write. Your ideas aren't obscured by boilerplate code and endlessly repeated idioms. This makes your code easier to maintain because you don't have to wade through reams of code every time you need to make a change."

Although he's writing about Lisp, the same logic applies to writing pure functions.

Another way that pure functions make code easier to reason about won't be apparent when you're first getting started. It turns out that what *really* happens in FP applications is that (a) you write as much of the code as you can in a functional style, and then (b) you have other functions that reach out and interact with files, databases, web services, UIs, and so on — everything in the outside world.

The concept is that you have a "Pure Function" core, surrounded by impure functions that interact with the outside world, as shown in Figure 11.1.

Given this design, a great thing about Haskell in particular is that it provides a clean separation between pure and impure functions — so clean that you can tell by looking at a function's signature whether it is pure or impure. I discuss this more in the coming lessons, but for now, just know that developers have built libraries to bring this same benefit to Scala/FP applications.

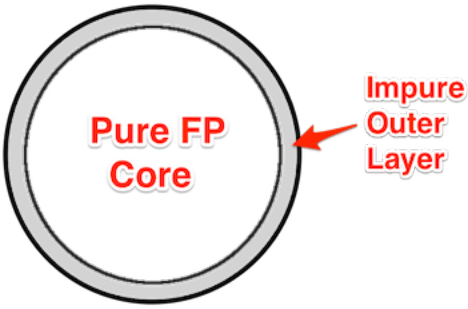


Figure 11.1: A pure FP core, with a thin layer of I/O functions.

2) Testing is easier, and pure functions lend themselves well to techniques like property-based testing

As I show in the Scala Cookbook, it's easier to test pure functions because you don't have to worry about them dealing with hidden state and side effects. What this means is that in imperative code you may have a method like this:

```
def doSomethingHidden(o: Order, p: Pizza): Unit ...
```

You can't tell much about what that method does by looking at its signature, but — because it returns nothing (`Unit`) — presumably it (a) modifies those variables, (b) changes some hidden state, or (c) interacts with the outside world.

When methods modify hidden state, you end up having to write long test code like this:

```
test("test hidden stuff that has side effects") {
    setUpPizzaState(p)
    setUpOrderState(o, p)
    doSomethingHidden(o, p)
    val result = getTheSideEffectFromThatMethod()
    assertEquals(result, expectedResult)
}
```

In FP you *can't* have code like that, so testing is simpler, like this:

```
test("test obvious stuff") {
    val result = doSomethingObvious(x, y, z)
    test(result, expectedResult)
}
```

Proofs

Beyond making unit testing easier, because functional code is like algebra it also makes it easier to use a form of testing known as property-based testing.

I write much more about this in the lesson on using ScalaCheck, but the main point is that because the outputs of your functions depend only on their inputs, you can define "properties" of your functions, and then ScalaCheck "attacks" your functions with a large range of inputs.

With a few minor edits, the property-based testing page on the ScalaTest website states:

> "… a *property* is a high-level specification of behavior that should hold for a range of data points. For example, a property might state, 'The size of a list returned from a method should always be greater than or equal to the size of the list passed to that method.' This property should hold no matter what list is passed."

> "The difference between a traditional unit test and a property is that unit tests traditionally verify behavior based on specific data points … for example, a unit test might pass three or four specific lists to a method that takes a list and check that the results are as expected. A property, by contrast, describes at a high level the preconditions of the method under test and specifies some aspect of the result that should hold no matter what valid list is passed."

3) Debugging is easier

> As Edsger Dijkstra said, "Program testing can be used to show the presence of bugs, but never to show their absence."

Because pure functions depend only on their input parameters to produce their output, debugging applications written with pure functions is easier. Of course it's possible to still make a mistake when you write a pure function, but once you have a stack trace or debug output, all you have to do is follow the values to see what went wrong. Because the functions are pure, you don't have to worry about what's going on in the rest of the application, you just have to know the inputs that were given to the pure function that failed.

In Masterminds of Programming, Paul Hudak, a co-creator of the Haskell language, states, "I've always felt that the 'execution trace' method of debugging in imperative languages was broken … in all my years of Haskell programming, I have never in fact used Buddha, or GHC's debugger, or any debugger at all … I find that testing works just fine; test small pieces of code using QuickCheck or a similar tool to make things more rigorous, and then — the key step — simply study the code to see why things don't work the way I expect them to. I suspect that a lot of people program similarly, otherwise there would be a lot more research on Haskell debuggers …"

> ScalaCheck is a property-based testing framework for Scala that was inspired by Haskell's QuickCheck.

4) Programs are more bulletproof

People that are smarter than I am can make the mathematical argument that *complete FP applications* are more bulletproof than other applications. Because there are fewer "moving parts" — mutatable variables and hidden state — in FP applications, mathematically speaking, the overall application is less complex. This is true for simple applications, and the gap gets larger in parallel and concurrent programming (as you'll see in a later section in this chapter).

The way I can explain this is to share an example of my own bad code. A few years ago I started writing a football game for Android devices (American football), and it has a lot of *state* to consider. On every play there is state like this:

- What quarter is it?
- How much time is left in the quarter?
- What is the score?
- What down is it?
- What distance is needed to make a first down?
- Much more …

Here's a small sample of the now-embarrassing `public static` fields I globally mutate in that application:

```
// stats for human
public static int numRunsByHuman            = 0;
public static int numPassAttemptsByHuman    = 0;
public static int numPassCompletionsByHuman = 0;
public static int numInterceptionsThrownByHuman = 0;
public static int numRunningYardsByHuman    = 0;
public static int numPassingYardsByHuman    = 0;
public static int numFumblesByHuman         = 0;
public static int numFirstDownRunsByHuman   = 0;
public static int numFirstDownPassesByHuman = 0;
```

When I wrote this code I thought, "I've written Java Swing (GUI) code since the 1990s, and Android code for a few years. I'm working by myself on this, I don't have to worry about team communication. I know what I'm doing, what could possibly go wrong?"

In short, although a football game is pretty simple compared to a business application, it still has a lot of "state" that you have to maintain. And when you're mutating that global state from several different places, well, it turns out that sometimes the computer gets an extra play, sometimes time doesn't run off the clock, etc.

Skipping all of my imperative state-related bugs … once I learned how to handle state in FP applications, I gave up trying to fix those bugs, and I'm now rewriting the core of the application in an FP style.

> As you'll see in this book, the solution to this problem is to pass the state around as a value, such as a case class or `Map`. In this case I might call it `GameState`, and it would have fields like `quarter`, `timeRemaining`, `down`, etc.

A second argument about FP applications being more bulletproof is that because they are built from all of these little pure functions that are known to work extraordinarily well, the overall application itself must be safer. For instance, if 80% of the application is written with well-tested pure functions, you can be very confident in that code; you know that it will never have the mutable state bugs like the ones in my football game. (And if somehow it does, the problem is easier to find and fix.)

As an analogy, one time I had a house built, and I remember that the builder was very careful about the 2x4's that were used to build the framework of the house. He'd line them up and then say, "You do *not* want that 2x4 in your house," and he would be pick up a bent or cracked 2x4 and throw it off to the side. In the same way that he was trying to build the framework of the house with wood that was clearly the best, we use pure functions to build the best possible core of our applications.

> Yes, I know that programmers don't like it when I compare building a house to writing an application. But some analogies do fit.

5) Programs are written at a higher level, and are therefore easier to comprehend

In the same way that pure functions are easier to reason about, overall FP applications are also easier to reason about. For example, I find that my FP code is more concise than my imperative and OOP code, and it's also still very readable. In fact, I think it's more readable than my older code.

Wikipedia states, "Imperative programming is a programming paradigm that uses statements that change a program's state."

Some of the features that make FP code more concise and still readable are:

- The ability to treat functions as values
- The ability to pass those values into other functions
- Being able to write small snippets of code as anonymous functions
- Not having to create deep hierarchies of classes (that sometimes feel "artifical")
- Most FP languages are "low ceremony" languages, meaning that they require less boilerplate code than other languages

  If you want to see what I mean by FP languages being "low ceremony," here's a good example of OCaml, and this page shows examples of Haskell's syntax.

In my experience, when I write Scala/FP code that I'm comfortable with today, I have always been able to read it at a later time. And as I mentioned when writing about the benefits of pure functions, "concise and readable" means that I can keep more code in my head at one time.

> I emphasize that Scala/FP code is *concise and readable* because sometimes "more concise" code can be a problem. I remember that a friend who didn't like Perl once described Perl code as, "Write once, read forever." Because the syntax could get so complex, he couldn't modify his own code a few weeks after writing it because he couldn't remember how each little syntactical nuance worked. I have the same problem writing complex regular expressions. If I don't document them when I create them, I can never tell how they work when I look at them later.
>
> (Personally I like the non-OO parts of Perl, and have written over 150 Perl tutorials.)

6) Pure function signatures are meaningful

When learning FP, another big "lightbulb going on over my head" moment came when I saw that my function signatures were suddenly much more meaningful than my imperative and OOP method signatures.

Because non-FP methods can have side effects — which are essentially *hidden* inputs and outputs of those methods — their function signatures often don't mean that much. For example, what do you think this imperative method does:

```
def doSomething(): Unit { code here ...
```

The correct answer is, *"Who knows?"* Because it takes no input parameters and returns nothing, there's no way to guess from the signature what this method does.

In contrast, because pure functions depend only on their input parameters to produce their output, their function signatures are extremely meaningful — a contract, even.

> I write more about this in the upcoming lesson, "Pure Functions Tell All."

7) Parallel programming

While writing parallel and concurrent applications is considered a "killer app" that helped spur renewed interest in FP, I have written my parallel/concurrent apps (like Sarah) primarily using Akka Actors and Scala Futures, so I can only speak about them: they're awesome tools. I wrote about them in the Scala Cookbook and on my website (alvinalexander.com), so please search those resources for "actors" and "futures" to find examples.

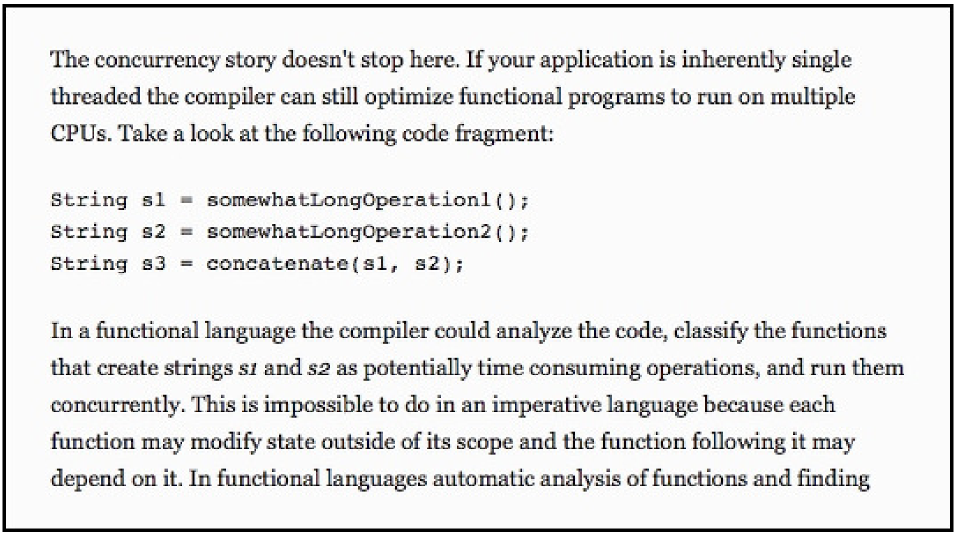
Therefore, to support the claims that FP is a great tool for writing parallel/concurrent applications, I'm going to include quotes here from other resources. As you'll see, the recurring theme in these quotes is, "Because FP only has immutable values, you can't possibly have the race conditions that are so difficult to deal with in imperative code."

The first quote comes from an article titled, "Functional Programming for the Rest of Us,":

> "A functional program is ready for concurrency without any further modifications. You never have to worry about deadlocks and race con-

> ditions because you don't need to use locks. No piece of data in a functional program is modified twice by the same thread, let alone by two different threads. That means you can easily add threads without ever giving conventional problems that plague concurrency applications a second thought."

The author goes on to add the information shown in Figure 11.2.



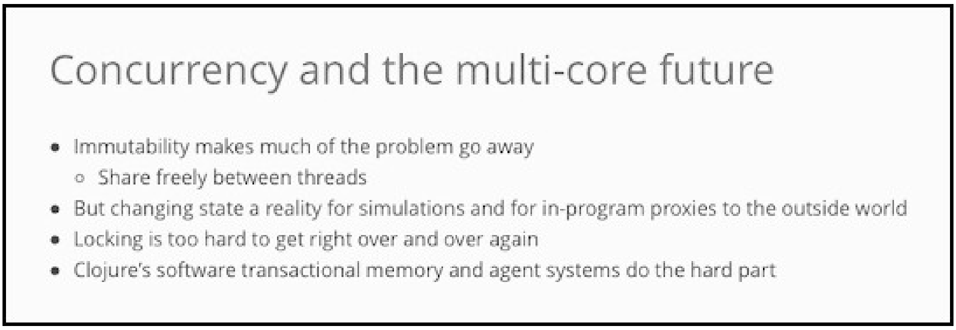
The concurrency story doesn't stop here. If your application is inherently single threaded the compiler can still optimize functional programs to run on multiple CPUs. Take a look at the following code fragment:

```
String s1 = somewhatLongOperation1();
String s2 = somewhatLongOperation2();
String s3 = concatenate(s1, s2);
```

In a functional language the compiler could analyze the code, classify the functions that create strings *s1* and *s2* as potentially time consuming operations, and run them concurrently. This is impossible to do in an imperative language because each function may modify state outside of its scope and the function following it may depend on it. In functional languages automatic analysis of functions and finding

Figure 11.2: A compiler can optimize functional programs to run on multiple cores.

The Clojure.org website adds the statements in Figure 11.3 about how Clojure and FP help with concurrency.



Concurrency and the multi-core future

- Immutability makes much of the problem go away
    - Share freely between threads
- But changing state a reality for simulations and for in-program proxies to the outside world
- Locking is too hard to get right over and over again
- Clojure's software transactional memory and agent systems do the hard part

Figure 11.3: Concurrency benefits from the Clojure website.

Page 17 of the book, Haskell, the Craft of Functional Programming, states, "Haskell programs are easy to parallelize, and to run efficiently on multicore hardware, because there is no state to be shared between different threads."

In this article on the ibm.com website, Neal Ford states, "Immutable objects are also automatically thread-safe and have no synchronization issues. They can also never exist in unknown or undesirable state because of an exception."

In the pragprom.com article, Functional Programming Basics, famous programmer Robert C. Martin extrapolates from four cores to a future with 131,072 processors when he writes:

> "Honestly, we programmers can barely get two Java threads to cooperate … Clearly, if the value of a memory location, once initialized, does not change during the course of a program execution, then there's nothing for the 131072 processors to compete over. You don't need semaphores if you don't have side effects! You can't have concurrent update problems if you don't update! … So that's the big deal about functional languages; and it is one big fricking deal. There is a freight train barreling down the tracks towards us, with multi-core emblazoned on it; and you'd better be ready by the time it gets here."

With a slight bit of editing, an article titled, The Downfall of Imperative Programming states:

> "Did you notice that in the definition of a *data race* there's always talk of mutation? Any number of threads may *read* a memory location without synchronization, but if even one of them *mutates* it, you have a race. And that is the downfall of imperative programming: Imperative programs will always be vulnerable to data races because they contain mutable variables."

id Software co-founder and technical director John Carmack states:

> "Programming in a functional style makes the state presented to your code explicit, which makes it much easier to reason about, and, in a completely pure system, makes thread race conditions impossible."

Writing Erlang code is similar to using the Akka actors library in Scala. The Erlang equivalent to an Akka actor is a "process," and in his book, Programming Erlang, Joe Armstrong writes:

> "Processes share no data with other processes. This is the reason why we can easily distribute Erlang programs over multicores or networks."

For a final quote, "The Trouble with Shared State" section on this medium.com article states, "In fact, if you're using shared state and that state is reliant on sequences which vary depending on indeterministic factors, for all intents and purposes, the output is impossible to predict, and that means it's impossible to properly test or fully understand. As Martin Odersky puts it:"

> non-determinism = parallel processing + mutable state

The author follows that up with an understatement: "Program determinism is usually a desirable property in computing."

Deterministic algorithms and concurrency

Deterministic algorithms

If you're not familiar with the term *deterministic algorithm*, Wikipedia defines it like this: "In computer science, a deterministic algorithm is an algorithm which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states."

(As you'll soon see, this is basically the definition of a pure function.)

Conversely, a *nondeterministic algorithm* is like asking a user to ask the person next to them what their favorite color is: you're never guaranteed to get the same answer. If you're trying to do something like sort a list of numbers, you *really* want a deterministic solution.

Parallel, Concurrent

Yossi Kreinin created the original version of the image shown in Figure 11.4 to help explain the differences between the meanings of “concurrent” and “parallel”.

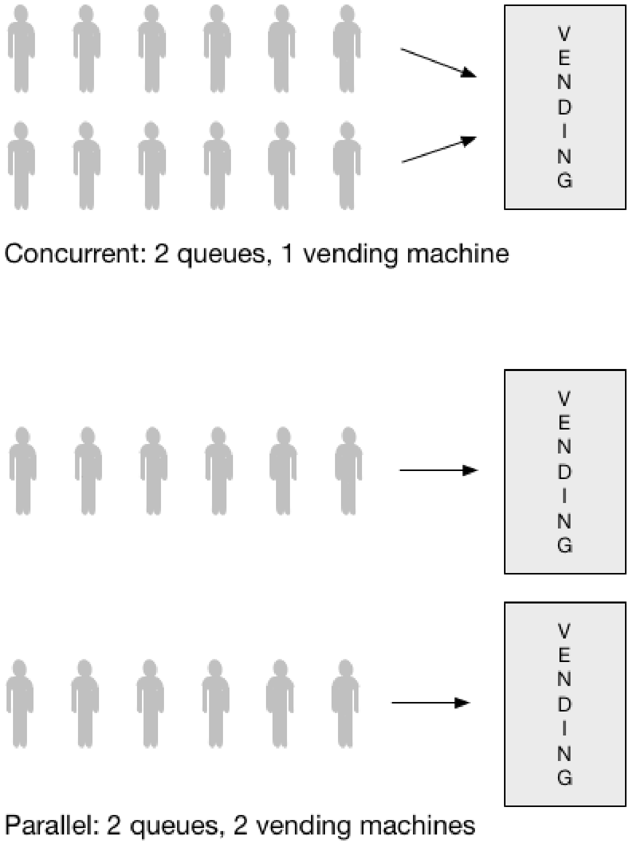


Figure 11.4: The difference between concurrent and parallel.

His image is based on a diagram in this article by famed Erlang programmer Joe Armstrong. Mr. Armstrong offers this summary in his post:

- Concurrent = Two queues and one coffee machine
- Parallel = Two queues and two coffee machines

I tend to use the two terms interchangeably, but I will be more precise with my language in the “Concurrency” lesson in this book.

8) Scala/FP benefit: The ability to treat functions as values

I've already written a little about higher-order functions (HOFs), and I write more about them later in this book, so I won't belabor this point: the fact that Scala (a) lets you treat functions as values, (b) lets you pass functions around to other functions, and (c) lets you write concise anonymous functions, are all features that make Scala a better functional programming language than another language (such as Java) that does not have these features.

9) Scala/FP benefit: Syntax makes function signatures easy to read

In my opinion, the Scala method syntax is about as simple as you can make method signatures, especially signatures that support generic types. This simplicity usually makes method signatures easy to read.

For instance, it's easy to tell that this method takes a `String` and returns an `Int`:

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

These days I prefer to use *explicit* return types on my methods, such as the `Int` in this example. I think that being explicit makes them easier to read later, when I'm in maintenance mode. And in an example like this, I don't know how to make that method signature more clear.

If you prefer methods with *implicit return types* you can write that same method like this, which is also clear and concise:

```
def foo(s: String) = ???
```

Even when you need to use generic type parameters — which make any method harder to read — Scala method signatures are still fairly easy to read:

```
def foo[A, B](a: A): B = ???
```

It's hard to make it much easier than that.

Occasionally I think that I'd like to get rid of the initial generic type declaration in the brackets — the `[A, B]` part — so the signature would look like this:

```
def foo(a: A): B = ???
```

While that's more concise for simple type declarations, it would create an inconsistency when you need to use advanced generic type features such as bounds and variance, like this example I included in the Scala Cookbook:

```
def getOrElse[B >: A](default: => B): B = ???
```

Even with the initial brackets, the type signatures are still fairly easy to read. You can make the argument that declaring the generic types in brackets before the rest of the signature makes it clear to the reader that they are about to see those types in the remainder of the signature. For instance, when you read this function:

```
def foo[A, B](a: A): B = ???
```

you can imagine saying to yourself, "This is a function named `foo` … its signature is going to use two generic types `A` and `B`, and then …"

Given what generic types represent, I think that's pretty clear.

> In Haskell, when you declare a function's type signature, you do it on a separate line, similar to the way that you declare C function signatures separately. For example, this is the way that you'd declare the signature for a Haskell function that takes an `Order` as an input parameter, and returns a `String` result:
>
> ```
> orderToString :: Order -&gt; String
> ```
>
> (Note: This is a simple example. One of the difficulties of learning Haskell is that its function signatures quickly get more complicated. See my Example Haskell pizza-ordering application for more function signature examples.)

10) Scala/FP benefit: The collections classes have a functional API

When I first came to Scala from Java, the Scala collections API was a real surprise. *But*, once I had that "Aha!" moment and realized how they work, I saw what a great benefit they are. Having all of those standard functional methods eliminates almost every need for custom `for` loops.

The important benefit of this is that these standard methods make my code more consistent and concise. These days I write almost 100% fewer custom `for` loops, and that's good for me — and anyone who has to read my code.

11) Scala/FP benefit: Code runs on the JVM

Because the Scala compiler generates Java bytecode that runs on the JVM, and because Scala supports both FP and OOP models, you can still use all of those thousands of Java/JVM libraries that have been created in the last twenty years in your Scala/FP applications. Even if those libraries aren't "Pure FP," at least you can still use them without having to "reinvent the wheel" and write a new library from scratch.

In fact, not only can you use the wealth of existing JVM libraries, you can also use all of your favorite JVM tools in general:

- Build tools like Ant, Maven, Gradle, and SBT
- Test tools like JUnit, TestNG, mock frameworks
- Continuous integration tools
- Debugging and logging frameworks
- Profiling tools
- More …

These libraries and tools are a great strength of the JVM. If you ask experienced FP developers why they are using Scala rather than Haskell or another FP language, "libraries, tools, and JVM" is the usual answer.

One more thing …

On a personal note, a big early influence for me — before I knew about any of these benefits — was seeing people like Martin Odersky, Jonas Bonér, Bill Venners, and other leading Scala programmers use and promote an FP style. Because Scala supports both OOP and FP, it's not like they had to sell anyone on FP in order to get us to use Scala. (As a former business owner, I feel like I'm always on the lookout for people who are trying to "sell" me something.)

I don't know if they use FP 100% of the time, but what influenced me is that they started using FP and then they never said, "You know what? FP isn't that good after all. I'm going back to an imperative style."

> In the 2016 version of Programming in Scala, Martin Odersky's biography states, "He works on programming languages and systems, more specifically on the topic of how to combine object-oriented and functional programming." Clearly FP is important to him (as is finding the best ways to merge FP and OOP concepts).

Summary

In summary, the benefits of "functional programming in general" are:

1. Pure functions are easier to reason about
2. Testing is easier, and pure functions lend themselves well to techniques like property-based testing
3. Debugging is easier
4. Programs are more bulletproof
5. Programs are written at a higher level, and are therefore easier to comprehend
6. Function signatures are more meaningful
7. Parallel/concurrent programming is easier

On top of those benefits, "functional programming in Scala" offers these additional benefits:

8. Being able to (a) treat functions as values and (b) use anonymous functions makes code more concise, and still readable
9. Scala syntax generally makes function signatures easy to read
10. The Scala collections' classes have a very functional API
11. Scala runs on the JVM, so you can still use the wealth of JVM-based libraries and tools with your Scala/FP applications

What's next

In this chapter I tried to share an honest assessment of the benefits of functional programming. In the next chapter I'll try to provide an honest assessment of the potential drawbacks and disadvantages of functional programming.

See Also

Quotes in this chapter came from the following sources:

- Real World Haskell
- Clean Code
- Masterminds of Programming
- Scala Cookbook
- The ScalaCheck website
- Property-based-testing on the ScalaTest website
- Functional Programming for the Rest of Us
- Yossi Kreinin's parallel vs concurrent image
- Joe Armstrong's parallel vs concurrent article
- The Clojure.org "rationale" page
- Haskell, the Craft of Functional Programming
- Neal Ford's comments on ibm.com

- Robert C. Martin’s Functional Programming Basics article
- The Downfall of Imperative Programming on fpcomplete.com
- The Erlang website
- The Akka website
- Programming Erlang
- If you want to take a look at OCaml, O’Reilly’s Real World OCaml is freely available online
- “The Trouble with Shared State” section of this medium.com article
- Deterministic algorithms on Wikipedia
- I found John Carmack’s quote in this reprinted article on gamasutra.com

You can also search my alvinalexander.com website for examples of Akka and Scala Futures.

12

Disadvantages of Functional Programming

> *"People say that practicing Zen is difficult, but there is a misunderstanding as to why."*
>
> Shunryu Suzuki,
> Zen Mind, Beginner's Mind

In the last chapter I looked at the benefits of functional programming, and as I showed, there are quite a few. In this chapter I'll look at the potential drawbacks of FP.

Just as I did in the previous chapter, I'll first cover the "drawbacks of functional programming *in general*":

1. Writing pure functions is easy, but combining them into a complete application is where things get hard.
2. The advanced math terminology (monad, monoid, functor, etc.) makes FP intimidating.
3. For many people, recursion doesn't feel natural.
4. Because you can't mutate existing data, you instead use a pattern that I call, "Update as you copy."
5. Pure functions and I/O don't really mix.
6. Using only immutable values and recursion can potentially lead to performance problems, including RAM use and speed.

After that I'll look at the more-specific "drawbacks of functional programming in Scala":

7. You can mix FP and OOP styles.
8. Scala doesn't have a standard FP library.

1) Writing pure functions is easy, but combining them into a complete application is where things get hard

Writing a pure function is generally fairly easy. Once you can define your type signature, pure functions are easier to write because of the absence of mutable variables, hidden inputs, hidden state, and I/O. For example, the `determinePossiblePlays` function in this code:

```
val possiblePlays = OffensiveCoordinator.determinePossiblePlays(gameState)
```

is a pure function, and behind it are thousands of lines of other functional code. Writing all of these pure functions took time, but it was never difficult. All of the functions follow the same pattern:

1. Data in
2. Apply an algorithm (to *transform* the data)
3. Data out

That being said, the part that *is* hard is, "How do I glue all of these pure functions together in an FP style?" That question can lead to the code I showed in the first chapter:

```
def updateHealth(delta: Int): Game[Int] = StateT[IO, GameState, Int]
    { (s: GameState) =>

    val newHealth = s.player.health + delta
      IO((s.copy(player = s.player.copy(health = newHealth)), newHealth))

}
```

As you may be aware, when you first start programming in a pure FP style, gluing pure functions together to create a complete FP application is one of the biggest stumbling blocks you'll encounter. In lessons later in this book I show solutions for how to glue pure functions together into a complete application.

2) Advanced math terminology makes FP intimidating

I don't know about you, but when I first heard terms like combinator, monoid, monad, and functor, I had no idea what people were talking about. And I've been paid to write software since the early-1990s.

As I discuss in the next chapter, terms like this are intimidating, and that "fear factor" becomes a barrier to learning FP.

> Because I cover this topic in the next chapter, I won't write any more about it here.

3) For many people, recursion doesn't feel natural

One reason I may not have known about those mathematical terms is because my degree is in aerospace engineering, not computer science. Possibly for the same reason, I *knew* about recursion, but never had to use it. That is, until I became serious about writing pure FP code.

As I wrote in the "What is FP?" chapter, the thing that happens when you use only pure functions and immutable values is that you *have* to use recursion. In pure FP code you no longer use `var` fields with `for` loops, so the only way to loop over elements in a collection is to use recursion.

Fortunately, you can learn how to write recursive code. If there's a secret to the process, it's in learning how to "think in recursion." Once you gain that mindset and see that there are patterns to recursive algorithms, you'll find that recursion gets much easier, even natural.

Two paragraphs ago I wrote, "the only way to loop over elements in a collection is to use recursion," but that isn't 100% true. In addition to gaining a "recursive thinking" mindset, here's another secret: once you understand the Scala collections' methods, you won't need to use recursion as often as you think. In the same way that collections' methods are replacements for custom `for` loops, they're also replacements for many custom recursive algorithms.

As just one example of this, when you first start working with Scala and you have a `List` like this:

```
val names = List("chris", "ed", "maurice")
```

it's natural to write a `for/yield` expression like this:

```
val capNames = for (e <- names) yield e.capitalize
```

As you'll see in the upcoming lessons, you can also write a recursive algorithm to solve this problem.

But once you understand Scala's collections' methods, you know that the `map` method is a replacement for those algorithms:

```
val capNames = fruits.map(_.e.capitalize)
```

Once you're comfortable with the collections' methods, you'll find that you reach for them before you reach for recursion.

> I write much more about recursion and the Scala collections' methods in upcoming lessons.

4) Because you can't mutate existing data, you instead use a pattern that I call, "Update as you copy"

For over 20 years I've written imperative code where it was easy — and extraordinarily common — to mutate existing data. For instance, once upon a time I had a niece named "Emily Means":

```
val emily = Person("Emily", "Means")
```

Then one day she got married and her last name became “Walls”, so it seemed logical to update her last name, like this:

```
emily.setLastName("Walls")
```

In FP you don’t do this. You don’t mutate existing objects.

Instead, what you do is (a) you copy an existing object to a new object, and then as a copy of the data is flowing from the old object to the new object, you (b) update any fields you want to change by providing new values for those fields, such as `lastName` in Figure 12.1.

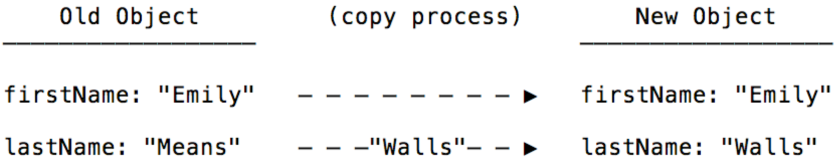


Figure 12.1: Results of the “update as you copy” concept

The way you “update as you copy” in Scala/FP is with the `copy` method that comes with *case classes*. First, you start with a case class:

```
case class Person (firstName: String, lastName: String)
```

Then, when your niece is born, you write code like this:

```
val emily1 = Person("Emily", "Means")
```

Later, when she gets married and changes her last name, you write this:

```
val emily2 = emily1.copy(lastName = "Walls")
```

After that line of code, `emily2.lastName` has the value `"Walls"`.

Note: I intentionally use the variable names `emily1` and `emily2` in this example to make it clear that you never change the original variable. In FP you constantly create intermediate variables like `name1` and `name2` during the “update as you copy” process, but there are FP techniques that make those intermediate variables transparent.

I show those techniques in upcoming lessons.

"Update as you copy" gets worse with nested objects

The "Update as you copy" technique isn't too hard when you're working with this simple `Person` object, but think about this: What happens when you have nested objects, such as a `Family` that has a `Person` who has a `Seq[CreditCard]`, and that person wants to add a new credit card, or update an existing one? (This is like an Amazon Prime member who adds a family member to their account, and that person has one or more credit cards.) Or what if the nesting of objects is even deeper?

In short, this is a real problem that results in some nasty-looking code, and it gets uglier with each nested layer. Fortunately, other FP developers ran into this problem long before I did, and they came up with ways to make this process easier.

I cover this problem and its solution in several lessons later in this book.

5) Pure functions and I/O don't really mix

As I wrote in the "What is Functional Programming" lesson, a *pure function* is a function (a) whose output depends only on its input, and (b) has no side effects. Therefore, by definition, any function that deals with these things is *impure*:

- File I/O
- Database I/O
- Internet I/O
- Any sort of UI/GUI input
- Any function that mutates variables
- Any function that uses "hidden" variables

Given this situation, a great question is, "How can an FP application possibly work without these things?"

The short answer is what I wrote in the Scala Cookbook and in the previous lesson: you write as much of your application's code in an FP style as you can, and then you write a thin I/O layer around the outside of the FP code, like putting "I/O icing" around an "FP cake," as shown in Figure 12.2.

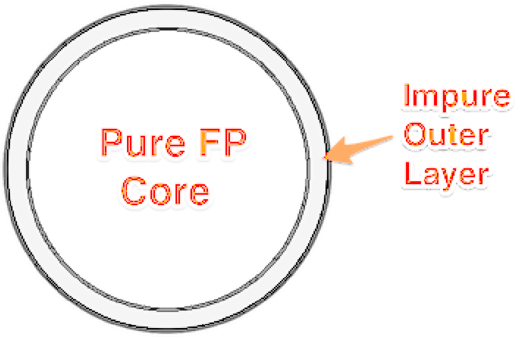


Figure 12.2: A thin, impure I/O layer around a pure core

Pure and impure functions

In reality, no programming language is really "pure," at least not by my definition. (Several FP experts say the same thing.) Wikipedia lists Haskell as a "pure" FP language, and the way Haskell handles I/O equates to this Scala code:

```
def getCurrentTime(): IO[String] = ???
```

The short explanation of this code is that Haskell has an `IO` type that you *must* use as a wrapper when writing I/O functions. This is enforced by the Haskell compiler.

For example, `getLine` is a Haskell function that reads a line from STDIN, and returns a type that equates to `IO[String]` in Scala. Any time a Haskell function returns something wrapped in an `IO`, like `IO[String]`, that function can only be used in certain places within a Haskell application.

> If that sounds hard core and limiting, well, it is. But it turns out to be a good thing.

Some people imply that this `IO` wrapper makes those functions pure, but in my opinion, this isn't true. At first I thought I was confused about this — that I didn't understand something — and then I read this quote from Martin Odersky on scala-lang.org:

> "The IO monad does not make a function pure. It just makes it obvious that it's impure."

For the moment you can think of an `IO` instance as being like a Scala `Option`. More accurately, you can think of it as being an `Option` that always returns a `Some[YourDataTypeHere]`, such as a `Some[Person]` or a `Some[String]`.

As you can imagine, just because you wrap a `String` that you get from the outside world inside of a `Some`, that doesn't mean the `String` won't vary. For instance, if you prompt me for my name, I might reply "Al" or "Alvin," and if you prompt my niece for her name, she'll reply "Emily," and so on. I think you'll agree that `Some["Al"]`, `Some["Alvin"]`, and `Some["Emily"]` are different values.

Therefore, even though (a) the return type of Haskell I/O functions must be wrapped in the `IO` type, and (b) the Haskell compiler only permits `IO` types to be in certain places, they are *impure* functions: they can return a different value each time they are called.

The benefit of Haskell's `IO` *type*

It's a little early in this book for me to write about all of this, but … the main benefit of the Haskell `IO` approach is that it creates a clear separation between (a) pure functions and (b) impure functions. Using Scala to demonstrate what I mean, I can look at this function and *know* from its signature that it's pure function:

```
def foo(a: String): Int = ???
```

Similarly, when I see that this next function returns something in an `IO` wrapper, I *know* from its signature alone that it's an impure function:

```
def bar(a: String): IO[String] = ???
```

That's actually very cool, and I write more about this in the I/O lessons of this book.

> I haven't discussed UI/GUI input/output in this section, but I discuss it more in the "Should I use FP everywhere?" section that follows.

6) Using only immutable values and recursion can lead to performance problems, including RAM use and speed

An author can get himself into trouble for stating that one programming paradigm can use more memory or be slower than other approaches, so let me begin this section by being very clear:

> *When you first write a simple ("naive") FP algorithm*, it is possible — just possible — that the immutable values and data-copying I mentioned earlier can be a performance problem.

I demonstrate an example of this problem in a blog post on Scala Quicksort algorithms. In that article I show that the basic ("naive") recursive `quickSort` algorithm found in the "Scala By Example" PDF uses about 660 MB of RAM while sorting an array of ten million integers, and is four times slower than using the `scala.util.Sorting.quickSort` method.

Having said that, it's important to note how `scala.util.Sorting.quickSort` works. In Scala 2.12, it passes an `Array[Int]` directly to `java.util.Arrays.sort(int[])`. The way that `sort` method works varies by Java version, but Java 8 calls a `sort` method in `java.util.DualPivotQuicksort`. The code in that method (and one other method it calls) is at least 300 lines long, and is *much* more complex than the simple/naive `quickSort` algorithm I show.

Therefore, while it's true that the "simple, naive" `quickSort` algorithm in the "Scala By Example" PDF has those performance problems, I need to be clear that I'm comparing (a) a very simple algorithm that you might initially write, to (b) a much larger, performance-optimized algorithm.

In summary, while this is a *potential* problem with simple/naive FP code, I offer solutions to these problems in a lesson titled, “Functional Programming and Performance.”

7) Scala/FP drawback: You can mix FP and OOP styles

If you’re an FP purist, a drawback to using *functional programming in Scala* is that Scala supports both OOP and FP, and therefore it’s possible to mix the two coding styles in the same code base.

While that is a potential drawback, many years ago when working with a technology known as Function Point Analysis — totally unrelated to functional programming — I learned of a philosophy called “House Rules” that eliminates this problem. With House Rules, the developers get together and agree on a programming style. Once a consensus is reached, that’s the style that you use. Period.

As a simple example of this, when I owned a computer programming consulting company, the developers wanted a Java coding style that looked like this:

```
public void doSomething()
{
    doX();
    doY();
}
```

As shown, they wanted curly braces on their own lines, and the code was indented four spaces. I doubt that everyone on the team loved that style, but once we agreed on it, that was it.

I think you can use the House Rules philosophy to state what parts of the Scala language your organization will use in your applications. For instance, if you want to use a strict “Pure FP” style, use the rules I set forth in this book. You can always change the rules later, but it’s important to start with something.

> There are two ways to look at the fact that Scala supports both OOP and FP. As mentioned, in the first view, FP purists see this as a drawback.
>
> But in a second view, people interested in using both paradigms within one language see this as a benefit. For example, Joe Armstrong has written that Erlang processes — which are the equivalent of Akka actors — can be written in an imperative style. Messages between processes are immutable, but the code within each process is single-threaded and can therefore be imperative. If a language only supports FP, the code in each process (actor) would have to be pure functional code, when that isn't strictly necessary.
>
> As I noted in the previous chapter, in the 2016 version of Programming in Scala, Martin Odersky's biography states, "He works on programming languages and systems, more specifically on the topic of how to combine object-oriented and functional programming." Trying to merge the two styles appears to be an important goal for Mr. Odersky.
>
> Personally, I like Scala's support of both the OOP and FP paradigms because this lets me use whatever style best fits the problem at hand. (In a terrific addition to this, adding Akka to the equation lets me use Scala the way other programmers use Erlang.)

8) Scala/FP drawback: Scala doesn't have a standard FP library

Another potential drawback to *functional programming in Scala* is that there isn't a built-in library to support certain FP techniques. For instance, if you want to use an `IO` data type as a wrapper around your impure Scala/FP functions, there isn't one built into the standard Scala libraries.

To deal with this problem, independent libraries like Scalaz, Cats, and others have been created. But, while these solutions are built into a language like Haskell, they are standalone libraries in Scala.

> I found that this situation makes it more difficult to learn Scala/FP. For instance, you can open any Haskell book and find a discussion of the `IO` type and other built-in language features, but the same is not true for

Scala. (I discuss this more in the I/O lessons in this book.)

> I considered comparing Scala's syntax to Haskell and other FP languages like F#/OCaml to demonstrate potential benefits and drawbacks, but that sort of discussion tends to be a personal preference: one developer's "concise" is another developer's "cryptic."
>
> If you want to avoid that sort of debate and read an objective comparison of Haskell and Scala features, Jesper Nordenberg provides one of the most neutral "Haskell vs Scala" discussions I've read.

"Should I use FP *everywhere*?"

Caution: A problem with releasing a book a few chapters at a time is that the later chapters that you'll finish writing at some later time can have an impact on earlier content. For this book, that's the case regarding this section. I have only worked with small examples of Functional Reactive Programming to date, so as I learn more about it, I expect that new knowledge to affect the content in this section. Therefore, a caution: "This section is still under construction, and may change significantly."

After I listed all of the benefits of functional programming in the previous chapter, I asked the question, "Should I write *all* of my code in an FP style?" At that time you might have thought, "Of course! This FP stuff sounds great!"

Now that you've seen some of the drawbacks of FP, I think I can provide a better answer.

1a) GUIs and Pure FP are not a good fit

The first part of my answer is that I like to write Android apps, and I also enjoy writing Java Swing and JavaFX code, and the interface between (a) those frameworks and (b) your custom code isn't a great fit for FP.

As one example of what I mean, in an Android football game I work on in my spare time, the OOP game framework I use provides an `update` method that I'm supposed to override to update the screen:

```
@Override
public void update(GameView gameView) {
    // my custom code here ...
}
```

Inside that method I have a lot of imperative GUI-drawing code that currently creates the UI shown in Figure 12.3.

There isn't a place for FP code at this point. The framework expects me to update the pixels on the screen within this method, and if you've ever written anything like a video game, you know that to achieve the best performance — and avoid screen flickering — it's generally best to update only the pixels that need to be changed. So this really is an "update" method, as opposed to a "completely redraw the screen" method.

> Remember, words like "update" and "mutate" are not in the FP vocabulary.

Other "thick client," GUI frameworks like Swing and JavaFX have similar interfaces, where they are OOP and imperative by design. Figure 12.4 shows an example of a little text editor I wrote and named "AlPad," and its major feature is that it lets me easily add and remove tabs to keep little notes organized.

The way you write Swing code like this is that you first create a `JTabbedPane`:

```
JTabbedPane tabbedPane = new JTabbedPane();
```

Once created, you keep that tabbed pane alive for the entire life of the application. Then when you later want to add a new tab, you *mutate* the `JTabbedPane` instance like this:

1st & 10 at 44

You: 0
Computer: 0

1st
11:48

1st & 10, 44

O: Screen pass
D: Probable blitz

7 rushes, 24 yards; 0-1-0, 0 yards

Figure 12.3: The UI for my "XO Play" application

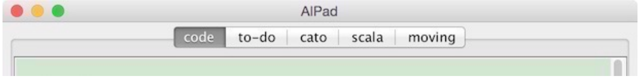


Figure 12.4: A few tabs in my "AlPad" application

```
tabbedPane.addTab(
    "to-do",
    null,
    newPanel,
    "to-do");
```

That's the way thick client code usually works: you create components and then mutate them during the life of the application to create the desired user interface. The same is true for most other Swing components, like `JFrame`, `JList`, `JTable`, etc.

Because these frameworks are OOP and imperative by nature, this interface point is where FP and pure functions typically don't fit.

> If you know about Functional Reactive Programming (FRP), please stand by; I write more on this point shortly.

When you're working with these frameworks you have to conform to their styles at this interface point, but there's nothing to keep you from writing the rest of your code in an FP style. In my Android football game I have a function call that looks like this:

```
val possiblePlays = OffensiveCoordinator.determinePossiblePlays(gameState)
```

In that code, `determinePossiblePlays` is a pure function, and behind it are several thousand lines of other pure functions. So while the GUI code has to conform to the Android game framework I'm using, the decision-making portion of my app — the "business logic" — is written in an FP style.

1b) Caveats to what I just wrote

Having stated that, let me add a few caveats.

First, *Web* applications are completely different than *thick client* (Swing, JavaFX) applications. In a thick client project, the entire application is typically written in one large codebase that results in a binary executable that users install on their computers. Eclipse, IntelliJ IDEA, and NetBeans are examples of this.

Conversely, the web applications I've written in the last few years use (a) one of many JavaScript-based technologies for the UI, and (b) the Play Framework on the server side. With Web applications like this, you have impure data coming into your Scala/Play application through data mappings and REST functions, and you probably also interact with impure database calls and impure network/internet I/O, but just like my football game, the "logic" portion of your application can be written with pure functions.

Second, the concept of Functional-Reactive Programming (FRP) combines FP techniques with GUI programming. The RxJava project includes this description:

> "RxJava is a Java VM implementation of Reactive Extensions: a library for composing asynchronous and event-based programs by using observable sequences … It extends the Observer Pattern to support sequences of data/events and adds operators that allow you to compose sequences together *declaratively* while abstracting away concerns about things like low-level threading, synchronization, thread-safety and concurrent data structures."
>
> (Note that declarative programming is the opposite of imperative programming.)

The ReactiveX.io website states:

> "ReactiveX is a combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming."

I provide some FRP examples later in this book, but this short example from the RxScala website gives you a taste of the concept:

```
object Transforming extends App {

    /**
     * Asynchronously calls 'customObservableNonBlocking'
     * and defines a chain of operators to apply to the
     * callback sequence.
     */
    def simpleComposition()
    {
        AsyncObservable.customObservableNonBlocking()
            .drop(10)
            .take(5)
            .map(stringValue => stringValue + "_xform")
            .subscribe(s => println("onNext => " + s))
    }

    simpleComposition()

}
```

This code does the following:

1. Using an "observable," it receives a stream of `String` values. Given that stream of values, it …
2. Drops the first ten values
3. "Takes" the next five values
4. Appends the string `"_xform"` to the end of each of those five values
5. Outputs those resulting values with `println`

As this example shows, the code that receives the stream of values is written in a functional style, using methods like `drop`, `take`, and `map`, combining them into a chain of calls, one after the other.

> I cover FRP in a lesson later in this book, but if you'd like to learn more now, the RxScala project is located here, and Netflix's "Reactive Programming in the Netflix API with RxJava" blog post is a good start.

This Haskell.org page shows current work on creating GUIs using FRP. (I'm not an expert on these tools, but at the time of this writing, most of these tools appear to be experimental or incomplete.)

2) Pragmatism (the best tool for the job)

I tend to be a pragmatist more than a purist, so when I need to get something done, I want to use the best tool for the job.

For instance, when I first started working with Scala and needed a way to stub out new SBT projects, I wrote a Unix shell script. Because this was for my personal use and I only work on Mac and Unix systems, creating a shell script was *by far* the simplest way to create a standard set of subdirectories and a *build.sbt* file.

Conversely, if I also worked on Microsoft Windows systems, or if I had been interested in creating a more robust solution like the Lightbend Activator, I might have written a Scala/FP application, but I didn't have those motivating factors.

> Another way to think about this is instead of asking, "Is FP the right tool for every application I need to write?," go ahead and ask that question with a different technology. For instance, you can ask, "Should I use *Akka actors* to write every application?" If you're familiar with Akka, I think you'll agree that writing an Akka application to create a few subdirectories and a *build.sbt* file would be overkill — even though Akka is a terrific tool for other applications.

Summary

In summary, potential drawbacks of *functional programming in general* are:

1. Writing pure functions is easy, but combining them into a complete application is where things get hard.
2. The advanced math terminology (monad, monoid, functor, etc.) makes FP intimidating.
3. For many people, recursion doesn't feel natural.
4. Because you can't mutate existing data, you instead use a pattern that I call, "Update as you copy."
5. Pure functions and I/O don't really mix.
6. Using only immutable values and recursion can potentially lead to performance problems, including RAM use and speed.

Potential drawbacks of *functional programming in Scala" are:

7. You can mix FP and OOP styles.
8. Scala doesn't have a standard FP library.

What's next

Having covered the benefits and drawbacks of functional programming, in the next chapter I want to help "free your mind," as Morpheus might say. That chapter is on something I call, "The Great FP Terminology Barrier," and how to break through that barrier.

See also

- My Scala Quicksort algorithms blog post
- Programming in Scala
- Jesper Nordenberg's "Haskell vs Scala" post
- Information about my "AlPad" text editor

- "Reactive Extensions" on reactivex.io
- Declarative programming
- Imperative programming
- The RxScala project
- Netflix's "Reactive Programming in the Netflix API with RxJava" blog post
- Functional Reactive Programming on haskell.org
- Lightbend Activator

13

The “Great FP Terminology Barrier”

> *“They say no ship can survive this.”*
>
> Hikaru Sulu, talking about “The Great Barrier” in *Star Trek V: The Final Frontier*

A short excursion to … The Twilight Zone

Hello, Rod Serling of The Twilight Zone here. Al will be back shortly, but for now, let me take you to another place and time … an alternate universe …

In this alternate universe you are born a few years earlier, and one day you find yourself writing some code. One week, you create a `List` class, and then a few days after that you find yourself writing the same `for` loops over and over again to iterate over list elements. Recognizing a pattern and also wanting to be DRY (“Don’t Repeat Yourself”), you create a cool new method on the `List` class to replace those repetitive `for` loops:

```
val xs = List(1, 2, 3).applyAFunctionToEveryElement(_ * 2)
```

You originally named this method, “apply a function to every element *and return a value for each element*,” but after deciding that was way too long for a function name, you shortened it to `applyAFunctionToEveryElement`.

But the problem with this shorter name is that it’s not technically accurate. Because you are applying a function to each element and then returning the corresponding result for each element, you need a better name. But what name is accurate — and concise?

Pulling out your handy thesaurus, you come up with possible method names like these:

- apply
- convert
- evolve
- transform
- transmute
- metamorphose

As you try to settle on which of these names is best, your mathematics buddy peers over your shoulder and asks, “What are you doing?” After you explain what you’re working on, he says, “Oh, cool. In mathematics we call that sort of thing ‘map.’ ” Then he pats you on the back, wishes you luck, and goes back to doing whatever it is that mathematicians do.

While some of the names you’ve come up with are good, this brief talk with your friend makes you think that it might be good to be consistent with mathematics. After all, you want mathematicians and scientists to use your programming language, so you decide to name your new method `map`:

```
val xs = List(1, 2, 3).map(_ * 2)
```

“Whoa,” you think to yourself, “that looks cool. I’ll bet there are zillions of functions that people can pass into `map` to achieve all kinds of cool things. And then I can use phrases like ‘map over a list.’” Things are taking shape.

`map` as a general concept

As you think about your invention, it occurs to you that there are at least a few different data types in the world that can be mapped over … not just lists, but hashmaps, too. Shoot, you can even think of a `String` as a `Seq[Char]`, and then even that can be mapped over. In time you realize that *any* collection whose elements can be iterated over can implement your new `map` function.

As this thought hits you, you realize that a logical thing to do is to create a trait that declares a `map` method. Then all of these other collections can extend that trait and implement their own `map` methods. With this thought, you begin sketching a new trait:

```
trait ThingsThatCanBeMappedOver {

    // extending classes need to implement this
    def map[A, B](f: A => B): TODO[B]

}
```

You realize that the `map` function signature isn't quite right — you're going to have to invent some other things to make this work — but never mind those details for now, you're on a roll.

With that trait, you can now implement your `List` class like this:

```
class List extends ThingsThatCanBeMappedOver {
    ...
}
```

As you write that first line of code you realize that the trait name `ThingsThatCanBeMappedOver` isn't quite right. It's accurate, but a little long and perhaps unprofessional. You start to pull out your thesaurus again, but that act makes you think of your math buddy; what would he call this trait?

It occurs to you that he would be comfortable writing code like this:

```
class List extends Map {
    ...
}
```

and as a result, you decide to call your new trait `Map`:

```
trait Map {

    // extending classes need to implement this
    def map[A, B](f: A => B): TODO[B]

}
```

There, that looks professional, and math-y like, too. Now you just have to figure out the correct function signature, and possibly implement a default method body.

Sadly, just at that moment, Rod Serling returns you to this version of planet Earth …

And the moral is …

In this version of Earth's history, someone beat you to the invention of "things that can be mapped over," and for some reason — possibly because they had a mathematics background — they made this declaration:

> "Things that can be mapped over shall be called … *Functor*."

Huh?

History did not record whether the Ballmer Peak, caffeine, or other chemicals were involved in that decision.

In this book, when I use the phrase, "Functional Programming Terminology Barrier," this is the sort of thing I'm referring to. If a normal human being had discovered this technique, they might have come up with a name like `ThingsThatCanBeMappedOver`, but a mathematician discovered it and came up with the name, "Functor."

> Moral: A lot of FP terminology comes from mathematics. Don't let it get you down.

A few more FP terms

As a few more examples of the *terminology barrier* I'm referring to, here are some other terms you'll run into as you try to learn functional programming:

Term	Definition
combinator	Per the Haskell wiki, this has two meanings, but the common meaning is, "a style of organizing libraries centered around the idea of combining things." This refers to being able to combine functions together like a Unix command pipeline, i.e., `ps aux | grep root | wc -l`
higher-order function	A function that takes other functions as parameters, or whose result is a function. (docs.scala-lang.org)
lambda	Another word for "anonymous function."

As these examples show, when you get into FP you'll start seeing new terminology, and oftentimes they aren't terms that you need to know for other forms of programming. For instance, I taught Java and OOP classes for five years, and I didn't know these words at that time. (As a reminder, my background is in aerospace engineering, not computer science.)

A common theme is that these terms generally come from mathematics fields like category theory. Personally, I like math, so this is good for me. When someone uses a term like "Combinatory Logic," I think, "Oh, cool, what's that? Is it something that can make me a better programmer?"

However, a bad thing about it is that it's easy to get lost in the terminology. If you've ever been lost in a forest, the feeling is just like that.

> As I write later in this book, I personally wasted a lot of time wondering, "What is *currying*? Why does everyone write about it so much?" That was a real waste of time.

I'll say this more than once in this book: the best thing you can do to learn FP is to write code using only pure functions and immutable values, and see where that leads you. I believe that if you place those restrictions on yourself, you'd eventually come up with the same inventions that mathematicians have come up with — and you might have simpler names for all of the terms.

> "Mathematicians have big, scary words like 'identity' and 'associativity' and 'commutativity' to talk about this stuff — it's their shorthand."
>
> ~ From the book, Coders at Work

More terms coming …

The key point of this lesson is that there's generally no need to worry about a lot of mathematical and FP jargon, especially when you're first getting started. As I found out through my own experience, all this terminology does is create a learning barrier.

That being said, one *good* thing about terminology is that it lets us know that we're all talking about the same thing. Therefore, I will introduce new terms as they naturally come up in the learning process. Which leads me to …

What's next

In the next lesson I'll formally define the term, "Pure Function." In this particular case — because I use the term so often throughout the remainder of the book, and it's a foundation of functional programming — it will help your learning process if I take a few moments to clearly define that term now.

See also

- The mathematical definition of “map” on Wikipedia
- The definition of “category theory” on Wikipedia
- If for some reason you want to see a lot of FP terms at this point, Cake Solutions has a nice Dictionary of functional programming
- Combinator on the Haskell Wiki
- Combinatory logic on the Haskell Wiki
- Combinator pattern on the Haskell Wiki
- Higher-order functions on scala-lang.org
- The Ballmer Peak
- The book, Coders at Work

14

Pure Functions

> *"When a function is pure, we say that 'output depends (only) on input.''*
>
> From the book, Becoming Functional
> (with the word "only" added by me)

Goals

This lesson has two goals:

1. Properly define the term "pure function."
2. Show a few examples of pure functions.

It also tries to simplify the pure function definition, and shares a tip on how to easily identify many impure functions.

Introduction

As I mentioned in the "What is Functional Programming?" chapter, I define functional programming (FP) like this:

> *Functional programming* is a way of writing software applications using only pure functions and immutable values.

Because that definition uses the term "pure functions," it's important to understand what a pure function is. I gave a partial pure function definition in that chapter, and now I'll provide a more complete definition.

Definition of “pure function”

Just like the term *functional programming*, different people will give you different definitions of a pure function. I provide links to some of those at the end of this lesson, but skipping those for now, Wikipedia defines a pure function like this:

1. The function always evaluates to the same result value given the same argument value(s). It cannot depend on any hidden state or value, and it cannot depend on any I/O.
2. Evaluation of the result does not cause any semantically observable side effect or output, such as mutation of mutable objects or output to I/O devices.

That’s good, but I prefer to reorganize those statements like this:

1. A pure function depends only on (a) its declared input parameters and (b) its algorithm to produce its result. A pure function has no “back doors,” which means:
    1. Its result can’t depend on *reading* any hidden value outside of the function scope, such as another field in the same class or global variables.
    2. It cannot *modify* any hidden fields outside of the function scope, such as other mutable fields in the same class or global variables.
    3. It cannot depend on any external I/O. It can’t rely on input from files, databases, web services, UIs, etc; it can’t produce output, such as writing to a file, database, or web service, writing to a screen, etc.
2. A pure function does not modify its input parameters.

This can be summed up concisely with this definition:

> A *pure function* is a function that depends *only* on its declared input parameters and its algorithm to produce its output. It does not read any other values from “the outside world” — the world outside of the function’s scope — and it does not modify any values in the outside world.

A mantra for writing pure functions

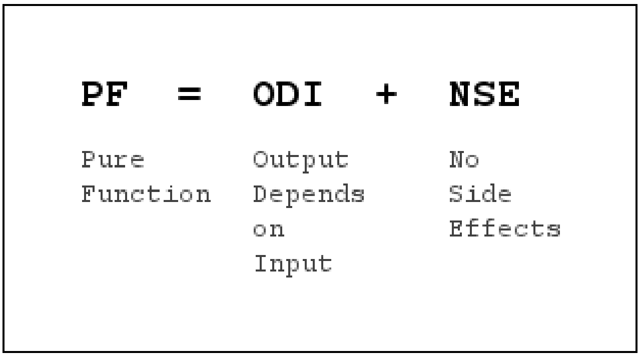
Once you’ve seen a formal pure function definition, I prefer this short mantra:

> Output depends *only* on input.

I like that because it’s short and easy to remember, but technically it isn’t 100% accurate because it doesn’t address side effects. A more accurate way of saying this is:

1. Output depends only on input
2. No side effects

You can represent that as shown in Figure 14.1.



PF = ODI + NSE

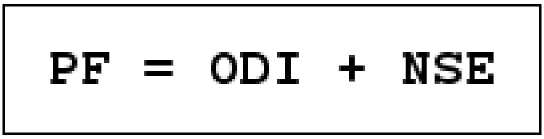
Pure
Function

Output
Depends
on
Input

No
Side
Effects

Figure 14.1: An equation to emphasize how pure functions work.

A simpler version of that equation is shown in Figure 14.2.



PF = ODI + NSE

Figure 14.2: A simpler version of that equation.

In this book I’ll generally either write, “Output depends on input,” or show one of these images.

The universe of a pure function

Another way to state this is that the universe of a pure function is only the input it receives, and the output it produces, as shown in Figure 14.3.

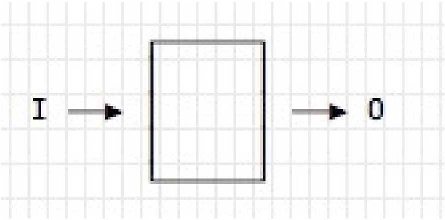


Figure 14.3: The entire universe of a pure function.

If it seems like I'm emphasizing this point a lot, it's because I am(!). One of the most important concepts of functional programming is that FP applications are built almost entirely with pure functions, and pure functions are very different than what I used to write in my OOP career. A great benefit of pure functions is that when you're writing them you don't have to think about anything else; all you have to think about is the universe of this function, what's coming in and what's going out.

Examples of pure and impure functions

Given the definition of pure functions and these simpler mantras, let's look at some examples of *pure* and *impure* functions.

Examples of pure functions

Mathematical functions are great examples of pure functions because it's pretty obvious that "output depends only on input." Methods like these in scala.math._ are all pure functions:

- `abs`
- `ceil`
- `max`

- `min`

I refer to these as "methods" because they are defined using `def` in the package object `math`. However, these methods work just like functions, so I also refer to them as pure functions.

Because a Scala `String` is immutable, every method available to a `String` is a pure function, including:

- `charAt`
- `isEmpty`
- `length`
- `substring`

Many methods that are available on Scala's collections' classes fit the definition of a pure function, including the common ones:

- `drop`
- `filter`
- `map`
- `reduce`

Examples of impure functions

Conversely, the following functions are *impure*.

Going right back to the collections' classes, the `foreach` method is impure. `foreach` is used only for its side effects, which you can tell by looking at its signature on the `Seq` class:

```
def foreach(f: (A) => Unit): Unit
```

Date and time related methods like `getDayOfWeek`, `getHour`, and `getMinute` are all impure because their output depends on something other than their inputs. Their results rely on some form of hidden I/O.

Methods on the `scala.util.Random` class like `nextInt` are also impure because their output depends on something other than their inputs.

In general, impure functions do one or more of these things:

- Read hidden inputs (variables not explicitly passed in as function input parameters)
- Write hidden outputs
- Mutate the parameters they are given
- Perform some sort of I/O with the outside world

Tip: Telltale signs of impure functions

By looking at function signatures *only*, there are two ways you can identify many impure functions:

- They don't have any input parameters
- They don't return anything (or they return `Unit` in Scala, which is the same thing)

For example, here's the signature for the `println` method of the Scala `Predef` object:

```
def println(x: Any): Unit
                     ----
```

Because `println` is such a commonly-used method, you already know that it writes information to the outside world, but if you didn't know that, its `Unit` return type would be a terrific hint of that behavior.

Similarly when you look at the "read*" methods that were formerly in `Predef` (and are now in scala.io.StdIn), you'll see that a method like `readLine` takes no input parameters, which is also a giveaway that it is impure:

```
def readLine(): String
            --
```

Because it takes no input parameters, the mantra, "Output depends only on input" clearly can't apply to it.

Simply stated:

- *If a function has no input parameters*, how can its output depend on its input?
- *If a function has no result*, it must have side effects: mutating variables, or performing some sort of I/O.

While this is an easy way to spot many impure functions, other impure methods can have both (a) input parameters and (b) a non-`Unit` return type, but still be impure because they read variables outside of their scope, mutate variables outside of their scope, or perform I/O.

Summary

As you saw in this lesson, this is my formal definition of a pure function:

> A *pure function* is a function that depends *only* on its declared inputs and its internal algorithm to produce its output. It does not read any other values from "the outside world" — the world outside of the function's scope — and it does not modify any values in the outside world.

Once you understand the complete definition, I prefer the short mantra:

> Output depends *only* on input.

or this more accurate statement:

1. Output depends only on input
2. No side effects

What's next

Now that you've seen the definition of a pure function, I'll show some problems that arise from using impure functions, and then summarize the benefits of using pure functions.

See also

- The Wikipedia definition of a pure function
- Wikipedia has a good discussion on "pure functions" on their Functional Programming page
- The wolfram.com definition of a pure function
- The schoolofhaskell.com definition of a pure function
- The ocaml.org definition of a pure function

15

Grandma’s Cookies (and Pure Functions)

To help explain pure functions, I’d like to share a little story …

Once upon a time I was a freshman in college, and my girlfriend’s grandmother sent her a tin full of cookies. I don’t remember if there were different kinds of cookies in the package or not — all I remember is the chocolate chip cookies. Whatever her grandmother did to make those cookies, the dough was somehow more white than any other chocolate chip cookie I had ever seen before. They also tasted terrific, and I ate most of them. (Sorry about that.)

Some time after this, my girlfriend — who would later become my wife — asked her grandmother how she made the chocolate chip cookies. Grandmother replied, “I just mix together some flour, butter, eggs, sugar, and chocolate chips, shape the dough into little cookies, and bake them at 350 degrees for 10 minutes.” (There were a few more ingredients, but I don’t remember them all.)

Later that day, my girlfriend and I tried to make a batch of cookies according to her grandmother’s instructions, but no matter how hard we tried, they always turned out like normal cookies. Somehow we were missing something.

Digging into the mystery

Perplexed by this mystery — and hungry for a great cookie — I snuck into grandmother’s recipe box late one night. Looking under “Chocolate Chip Cookies,” I found these comments:

```
/**
 * Mix together some flour, butter, eggs, sugar,
 * and chocolate chips. Shape the dough into
 * little cookies, and bake them at 350 degrees
 * for 10 minutes.
 */
```

"Huh," I thought, "that's just what she told us."

I started to give up on my quest after reading the comments, but the desire for a great cookie spurred me on. After thinking about it for a few moments, I realized that I could decompile grandmother's `makeCookies` recipe to see what it showed. When I did that, this is what I found:

```
def makeCookies(ingredients: List[Ingredient]): Batch[Cookie] = {
    val cookieDough = mix(ingredients)
    val betterCookieDough = combine(cookieDough, love)
    val cookies = shapeIntoLittleCookies(betterCookieDough)
    bake(cookies, 350.DegreesFahrenheit, 10.Minutes)
}
```

"Aha," I thought, "here's some code I can dig into."

Looking at the first line, the function declaration seems fine:

```
def makeCookies(ingredients: List[Ingredient]): Batch[Cookie] = {
```

Whatever `makeCookies` does, as long as it's a pure function — where its output depends only on its declared inputs — its signature states that it transforms a list of ingredients into a batch of cookies. Sounds good to me.

The first line inside the function says that `mix` is some sort of algorithm that transforms `ingredients` into `cookieDough`:

```
val cookieDough = mix(ingredients)
```

Assuming that `mix` is a pure function, this looks good.

The next line looks okay:

```
val betterCookieDough = combine(cookieDough, love)
```

Whoa. Hold on just a minute … now I'm confused. What is `love`? Where does `love` come from?

Looking back at the function signature:

```
def makeCookies(ingredients: List[Ingredient]): Batch[Cookie] = {
```

clearly `love` is not defined as a function input parameter. Somehow `love` snuck into this function. That's when it hit me:

> "Aha! `makeCookies` is not a pure function!""

Taking a deep breath to get control of myself, I looked at the last two lines of the function, and with the *now-major* assumption that `shapeIntoLittleCookies` and `bake` are pure functions, those lines look fine:

```
val cookies = shapeIntoLittleCookies(betterCookieDough)
bake(cookies, 350.DegreesFahrenheit, 10.Minutes)
```

"I don't know where `love` comes from," I thought, "but clearly, it is a problem."

Hidden inputs and free variables

In regards to the `makeCookies` function, you'll hear functional programmers say a couple of things about `love`:

- `love` is a *hidden input* to the function
- `love` is a "free variable"

These statements essentially mean the same thing, so I prefer the first statement: to think of `love` as being a hidden input into the function. It wasn't passed in as a function input parameter, it came from … well … it came from somewhere else … the ether.

Functions as factories

Imagine that `makeCookies` is the only function you have to write today — this function is *your* entire scope for today. When you do that, it feels like someone teleported `love` right into the middle of your workspace. There you were, minding your own business, writing a function whose output depends only on its inputs, and then — Bam! — `love` is thrown right into the middle of your work.

Put another way, if `makeCookies` is the entire scope of what you should be thinking about right now, using `love` feels like you just accessed a global variable, doesn't it?

With pure functions I like to think of input parameters as coming into a function's front door, and its results going out its back door, just like a black box, or a factory, as shown in Figure 15.1.

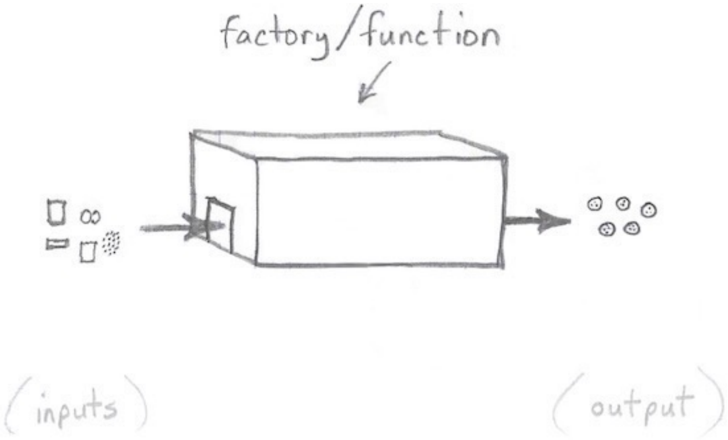


Figure 15.1: Thinking of a pure function as a factory with two doors.

But in the case of `makeCookies` it's as though `love` snuck in through a side door, as shown in Figure 15.2.

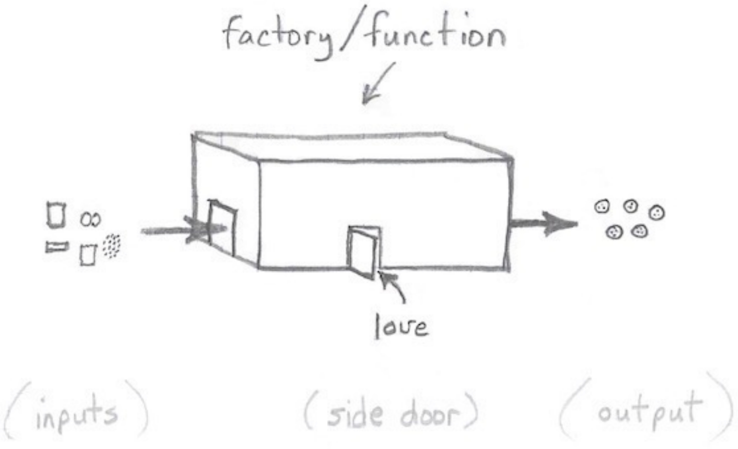


Figure 15.2: Impure functions use side doors.

While you might think it's okay for things like `love` to slip in a side door, if you spend any time in Alaska you'll learn not to leave your doors open, because you never know what might walk in, as shown in Figure 15.3.

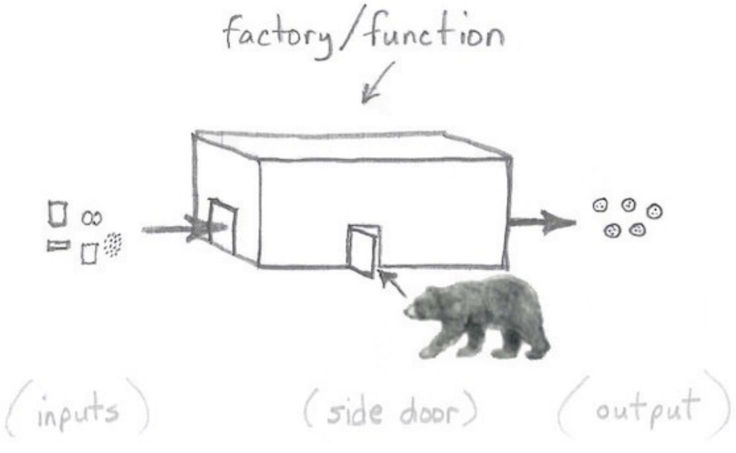


Figure 15.3: Bad things can happen when you use side doors.

Free variables

When I wrote about hidden inputs I also mentioned the term "free variable," so let's look at its meaning. Ward Cunningham's c2.com website defines a free variable like this:

> "A *free variable* is a variable used within a function, which is neither a formal parameter to the function nor defined in the function's body."

That sounds exactly like something you just heard, right? As a result, I prefer to use the less formal term, "hidden input."

What happens when hidden inputs change?

If Scala required us to mark impure functions with an `impure` annotation, `makeCookies` would be declared like this as a warning to all readers that, "Output depends on something other than input":

```
@impure
def makeCookies ...
```

And because `makeCookies` is an impure function, a good question to ask right now is:

"What happens when `love` changes?"

The answer is that because `love` comes into the function through a side door, it can change the `makeCookies` result without you ever knowing why you can get different results when you call it. (Or why my cookies never turn out right.)

Unit tests and purity

I like to "speak in source code" as much as possible, and a little code right now can show what a significant problem hidden inputs are, such as when you write a unit test for an impure method like `makeCookies`.

If you're asked to write a ScalaTest unit test for `makeCookies`, you might write some code like this:

```
test("make a batch of chocolate chip cookies") {
    val ingredients = List(
        Flour(3.Cups),
        Butter(1.Cup),
        Egg(2),
        Sugar(1.Cup),
        ChocolateChip(2.Cups)
    )
    val batchOfCookies = GrandmasRecipes.makeCookies(ingredients)
    assert(cookies.count == 12)
    assert(cookies.taste == Taste.JustLikeGrandmasCookies)
    assert(cookies.doughColor == Color.WhiterThanOtherCookies)
}
```

If you ran this test once it might work fine, you *might* get the expected results. But if you run it several times, you might get different results each time.

That's a big problem with `makeCookies` using `love` as a hidden input: when you're writing black-box testing code, you have no idea that `makeCookies` has a hidden dependency on `love`. All you'll know is that sometimes the test succeeds, and other times it fails.

Put a little more technically:

- `love`'s state affects the result of `makeCookies`
- As a black-box consumer of this function, there's no way for you to know that `love` affects `makeCookies` by looking at its method signature

If you have the source code for `makeCookies` and can perform white-box testing, you *can* find out that `love` affects its result, but that's a big thing about functional programming: you never have to look at the source code of a pure function to see if it has hidden inputs or hidden outputs.

> I've referred to hidden inputs quite a bit so far, but hidden outputs — mutating hidden variables or writing output — are also a problem of impure functions.

Problems of the impure world

However, now that I do have the `makeCookies` source code, several questions come to mind:

- Does `love` have a default value?
- How is `love` set before you call `makeCookies`?
- What happens if `love` is not set?

Questions like these are problems of *impure* functions in general, and *hidden inputs* in particular. Fortunately you don't have to worry about these problems when you write pure functions.

> When you write parallel/concurrent applications, the problem of hidden inputs becomes even worse. Imagine how hard it would be to solve the problem if `love` is set on a separate thread.

The moral of this story

Every good story should have a moral, and I hope you see what a problem this is. In my case, I still don't know how to make cookies like my wife's grandmother did. (I lay in bed at night wondering, what is `love`? Where does `love` come from?)

In terms of writing rock-solid code, the moral is:

- `love` is a hidden input to `makeCookies`
- `makeCookies` output does not depend solely on its declared inputs
- You may get a different result every time you call `makeCookies` with the same inputs
- You can't just read the `makeCookies` signature to know its dependencies

Programmers also say that `makeCookies` depends on the *state* of `love`. Furthermore, with this coding style it's also likely that `love` is a mutable `var`.

> My apologies to my wife's grandmother for using her in this example. She was the most organized person I ever met, and I'm sure that if she was a programmer, she would have written pure functions. And her cookies are sorely missed.

What's next

Given all of this talk about pure functions, the next lesson answers the important question, "What are the benefits of pure functions?"

See also

- The Wikipedia definition of a pure function
- Wikipedia has a good discussion on "pure functions" on their Functional Programming page
- My unit test was written using ScalaTest.
- When you need to use specific *quantities* in Scala applications, Squants offers a DSL similar to what I showed in these examples.

16

Benefits of Pure Functions

> *When asked, "What are the advantages of writing in a language without side effects?," Simon Peyton Jones, co-creator of Haskell, replied, "You only have to reason about values and not about state. If you give a function the same input, it'll give you the same output, every time. This has implications for reasoning, for compiling, for parallelism."*
>
> From the book, Masterminds of Programming

The goal of this lesson is simple: to list and explain the benefits of writing pure functions.

Benefits of pure functions

My favorite benefits of pure functions are:

- They're easier to reason about
- They're easier to combine
- They're easier to test
- They're easier to debug
- They're easier to parallelize

FP developers talk about other benefits of writing pure functions. For instance, Venkat Subramaniam adds these benefits:

- They are idempotent
- They offer referential transparency
- They are memoizable
- They can be lazy

In this lesson I'll examine each of these benefits.

Pure functions are easier to reason about

Pure functions are easier to reason about than impure functions, and I cover this in detail in the lesson, "Pure Function Signatures Tell All." The key point is that because a pure function has no side effects or hidden I/O, you can get a terrific idea of what it does just by looking at its signature.

Pure functions are easier to combine

Because "output depends only on input," pure functions are easy to combine together into simple solutions. For example, you'll often see FP code written as a chain of function calls, like this:

```
val x = doThis(a).thenThis(b)
                 .andThenThis(c)
                 .doThisToo(d)
                 .andFinallyThis(e)
```

This capability is referred to as *functional composition*. I'll demonstrate more examples of it throughout this book.

As you'll see in the "FP is Like Unix Pipelines" lesson, Unix pipelines work extremely well because most Unix commands are like pure functions: they read input and produce transformed output based only on the inputs and the algorithm you supply.

Pure functions are easier to test

As I showed in the "Benefits of Functional Programming" chapter and the unit test in the previous lesson, pure functions are easier to test than impure functions. I expand on this in several other lessons in this book, including the lesson on property-based testing.

Pure functions are easier to debug

In the "Benefits of Functional Programming" chapter I wrote that on a large scale, FP *applications* are easier to debug. In the small scale, pure functions are also easier to debug than their impure counterparts. Because the output of a pure function depends only on the function's input parameters and your algorithm, you don't need to look outside the function's scope to debug it.

Contrast that with having to debug the `makeCookies` function in the previous lesson. Because `love` is a hidden input, you have to look outside the function's scope to determine what `love`'s state was at the time `makeCookies` was called, and how that state was set.

Pure functions are easier to parallelize

In that same chapter I also wrote that it's easier to write parallel/concurrent applications with FP. Because all of those same reasons apply here I won't repeat them, but I will show one example of how a compiler can optimize code within a pure function.

I'm not a compiler writer, so I'll begin with this statement from the "pure functions" section of the Wikipedia functional programming page:

> "If there is no data dependency between two pure expressions, then their order can be reversed, or they can be performed in parallel and they cannot interfere with one another (in other terms, the evaluation of any pure expression is thread-safe)."

As an example of what that means, in this code:

```
val x = f(a)
val y = g(b)
val z = h(c)
val result = x + y + z
```

there are no data dependencies between the first three expressions, so they can be executed in any order. The only thing that matters is that they are executed before the assignment to `result`. If the compiler/interpreter wants to run those expressions in parallel, it can do that and then merge their values in the final expression. This can happen because (a) the functions are pure, and (b) there are no dependencies between the expressions.

That same Wikipedia page also states:

> "If the entire language does not allow side-effects, then any evaluation strategy can be used; this gives the compiler freedom to reorder or combine the evaluation of expressions in a program (for example, using deforestation)."

> The 2006 article, Functional Programming for the Rest Of Us, includes a quote similar to these Wikipedia quotes. It states, "An interesting property of functional languages is that they can be reasoned about mathematically. Since a functional language is simply an implementation of a formal system, all mathematical operations that could be done on paper still apply to the programs written in that language. The compiler could, for example, convert pieces of code into equivalent but more efficient pieces with a mathematical proof that two pieces of code are equivalent. Relational databases have been performing these optimizations for years. There is no reason the same techniques can't apply to regular software."

Pure functions are idempotent

I don't use the word "idempotent" too often, so I'll quote from Venkat Subramaniam's explanation of the benefit of idempotence in regards to pure functions (with a few minor edits by me):

> The word *idempotent* has a few different meanings … a function or operation is idempotent if the result of executing it multiple times for a given input is the same as executing it only once for the same input. If we know that an operation is idempotent, we can run it as many times as we like … it's safe to retry.

In a related definition, in A practical introduction to functional programming, Mary Rose Cook states:

> A process is *deterministic* if repetitions yield the same result every time.

The terms *idempotent* and *deterministic* are similar to a favorite phrase of mine: you can call a pure function an infinite number of times and always get the same result.

> Honestly, with these definitions it feels like I'm writing, "A benefit of pure functions is that they are pure functions." My only reason for keeping this section is so that you have some exposure to the terms *idempotent* and *deterministic*.
>
> This demonstrates that like many other uncommon phrases in functional programming, you can understand a *concept* long before you know that someone created a label for that concept.

Pure functions offer referential transparency

Referential transparency (RT) is another technical term that you'll hear in the FP world. It's similar to idempotency, and refers to what you (and a compiler) can do because your functions are pure.

If you like algebra, you'll like RT. It's said that an expression is *referentially transparent* if it can be replaced by its resulting value without changing the behavior of the program.

For instance, assume that `x` and `y` are immutable values within some scope of an application, and within that scope they're used to form this expression:

```
x + y
```

Then you can assign this expression to a third variable `z`:

```
val z = x + y
```

Now, throughout the given scope of your program, anywhere the expression `x + y` is used, it can be replaced by `z` without affecting the result of the program (and vice-versa).

Note that although I state that `x` and `y` are immutable values, they can also be the result of pure functions. For instance, `"hello".length + "world".length` will always be `10`. This result could be assigned to `z`, and then `z` could be used everywhere instead of this expression. In Scala this looks like this:

```
val x = "hello".length   // 5
val y = "world".length   // 5
val z = x + y            // 10
```

Because all of those values are immutable, you can use `z` anywhere you might use `x+y`, and in fact, in this example you can replace `z` with `10` anywhere, and your program will run exactly the same.

> In FP we say things like, "`10` cannot be reduced any more." (More on this later.)

Conversely, if `x` or `y` was an impure function, such as a "get the current time" function, `z` could not be a reliable replacement for `x + y` at different points in the application.

Pure functions are memoizable

Because a pure function always returns the same result when given the same inputs, a compiler (or your application) can also use caching optimizations, such as *memoization*.

Wikipedia defines memoization like this:

> "Memoization is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again."

For example, I previously noted that my Android football game has this function call:

```
val possiblePlays = OffensiveCoordinator.determinePossiblePlays(gameState)
```

The `determinePossiblePlays` function currently has several thousand lines of pure functions behind it, and over time it's only going to get more complicated. Although this function doesn't currently use memoization, it would be fairly simple to create a cache for it, so that each time it received the same `gameState` it would return the same result.

The cache could be implemented as a `Map`, with a type of `Map[GameState, Seq[OffensivePlay]]`. Then when `determinePossiblePlays` receives a `GameState` instance, it could perform a fast lookup in this cache.

> While those statements are true, I don't want to oversimplify this too much. `determinePossiblePlays` makes decisions based on many `GameState` factors, including two important (a) game score and (b) time remaining. Those two variables would have to be factors in any cache.

Pure functions can be lazy

Laziness is a major feature of the Haskell language, where everything is lazy. In Scala I primarily use laziness with large data sets and streams, so I haven't personally taken advantage of this benefit yet.

> (I'll update this benefit when I have a good Scala example.)

Summary

In this lesson I wrote about the benefits of pure functions. My favorite benefits are:

- They're easier to reason about
- They're easier to combine
- They're easier to test
- They're easier to debug
- They're easier to parallelize

Other FP developers write about these benefits of pure functions:

- They are idempotent
- They offer referential transparency
- They are memoizable
- They can be lazy

See also

- Wikipedia has a good discussion on the benefits of "pure functions" on their Functional Programming page
- The Haskell.org definition of referential transparency
- Stack Exchange provides a definition of referential transparency
- Stack Overflow says, Don't worry about the term RT, it's for pointy-headed purists
- Venkat Subramaniam's post on the benefits of pure functions
- If you like debates on the precise meaning of technical terms, reddit.com has a thread titled, Purity and referential transparency are different

17

Pure Functions and I/O

> *"The ancient Greeks have a knack of wrapping truths in myths."*
>
> George Lloyd

Goal

The goal of this lesson is to answer the question, "Because pure functions can't have I/O, how can an FP application possibly get anything done if all of its functions are pure functions?"

So how do you do anything with functional programming?

Given my pure function mantra, "Output depends only on input," a perfectly rational question at this point is:

> "How do I get anything done if I can't read any inputs or write any outputs?"

Great question!

The answer is that you violate the "Write Only Pure Functions" rule! It seems like other books go through great lengths to avoid answering that question until the final chapters, but I just gave you that answer fairly early in this book. (You're welcome.)

The general idea is that you write as much of your application as possible in an FP style, and then handle the UI and all forms of input/output (I/O) (such as Database

I/O, Web Service I/O, File I/O, etc.) in the best way possible for your current programming language and tools.

In Scala the percentage of your code that's considered impure I/O will vary, depending on the application type, but will probably be in this range:

- On the low end, it will be about the same as a language like Java. So if you were to write an application in Java and 20% of it was going to be impure I/O code and 80% of it would be other stuff, in FP that "other stuff" will be pure functions. This assumes that you treat your UI, File I/O, Database I/O, Web Services I/O, and any other conceivable I/O the same way that you would in Java, without trying to "wrap" that I/O code in "functional wrappers." (More on this shortly.)
- On the high end, it will approach 100%, where that percentage relies on two things. First, you wrap all of your I/O code in functional wrappers. Second, your definition of "pure function" is looser than the definition I have stated thus far.

I/O wrapper's code

I don't mean to make a joke or be facetious in that second statement. It's just that some people may try to tell you that by putting a wrapper layer around I/O code, the *impure* I/O function somehow becomes *pure*. Maybe somewhere in some mathematical sense that is correct, I don't know. Personally, I don't buy that.

Let me explain what I'm referring to.

Imagine that in Scala you have a function that looks like this:

```
def promptUserForUsername: String = ???
```

Clearly this function is intended to reach out into the outside world and prompt a user for a username. You can't tell *how* it does that, but the function name and the fact that it returns a `String` gives us that impression.

Now, as you might expect, every user of an application (like Facebook or Twitter) should have a unique username. Therefore, any time this function is called, it will return a different result. By stating that (a) the function gets input from a user, and (b) it can return a different result every time it's called, this is clearly not a *pure* function. It is *impure*.

However, now imagine that this same function returns a `String` that is wrapped in another class that I'll name `IO`:

```
def promptUserForUsername: IO[String] = ???
```

This feels a little like using the Option/Some/None pattern in Scala.

What's the benefit?

That's interesting, but what does this do for us?

Personally, I think it has one main benefit: I can glance at this function signature, and know that it deals with I/O, and therefore it's an impure function. In this particular example I can also infer that from the function name, but what if the function was named differently?:

```
def getUsername: IO[String] = ???
```

In this case `getUsername` is a little more ambiguous, so if it just returned `String`, I wouldn't know exactly how it got that `String`. But when I see that a `String` is wrapped with `IO`, I know that this function interacts with the outside world to get that `String`. That's pretty cool.

Does using `IO` *make a function pure?*

But this is where it gets interesting: some people state that wrapping `promptUserForUsername`'s return type with `IO` makes it a pure function.

I am not that person.

The way I look at it, the first version of `promptUserForUsername` returned `String` values like these:

```
"alvin"
"kim"
"xena"
```

and now the second version of `promptUserForUsername` returns that same infinite number of different strings, but they're wrapped in the `IO` type:

```
IO("alvin")
IO("kim")
IO("xena")
```

Does that somehow make `promptUserForUsername` a pure function? I sure don't think so. It still interacts with the outside world, and it can still return a different value every time it's called, so by definition it's still an impure function.

As Martin Odersky states in this Google Groups Scala debate:

> "The `IO` monad does not make a function pure. It just makes it obvious that it's impure."

Where does `IO` *come from?*

As I noted in the "What is This Lambda You Speak Of?" chapter, monads were invented in 1991, and added to Haskell in 1998, with the `IO` monad becoming Haskell's way of handling input/output. Therefore, I'd like to take a few moments to explain why this is such a good idea *in Haskell*.

I/O in Haskell

If you come from the Java world, the best thing you can do at this moment is to forget anything you know of how the Java Virtual Machine (JVM) works. By that, I mean that you should not attempt to apply anything you know about the JVM to what I'm about to write, because the JVM and Haskell compiler are as different as dogs and cats.

Haskell is considered a "pure" functional programming language, and when monads were invented in the 1990s, the `IO` monad became the Haskell way to handle I/O. In Haskell, any function that deals with I/O *must* declare its return type to be `IO`. *This is not optional.* Functions that deal with I/O *must* return the `IO` type, and this is enforced by the compiler.

For example, imagine that you want to write a function to read a user's name from the command line. In Haskell you'd declare your function signature to look like this:

```
getUsername :: IO String
```

In Scala, the equivalent function will have this signature:

```
def getUsername: IO[String] = ???
```

A great thing about Haskell is that declaring that a function returns something inside of an outer "wrapper" type of `IO` is a signal to the compiler that this function is going to interact with the outside world. As I've learned through experience, this is also a nice signal to other developers who need to read your function signatures, indicating, "This function deals with I/O."

There are two consequences of the `IO` type being a signal to the Haskell compiler:

1. The Haskell compiler is free to optimize any code that does not return something of type `IO`. This topic really requires a long discussion, but in short, the Haskell compiler is free to re-order all non-`IO` code in order to optimize it. Because pure functional code is like algebra, the compiler can treat all non-`IO` functions as mathematical equations. This is somewhat similar to how a relational database optimizes your queries. (That is a very short summary of a

large, complicated topic. I discuss this more in the "Functional Programming is Like Algebra" lesson.)

2. You can only use Haskell functions that return an `IO` type in certain areas of your code, specifically (a) in the `main` block or (b) in a `do` block. Because of this, if you attempt to use the `getUsername` function outside of a `main` or `do` block, your code won't compile.

If that sounds pretty hardcore, well, it is. But there are several benefits of this approach.

First, you can always tell from a function's return type whether it interacts with the outside world. Any time you see that a function returns something like an `IO[String]`, you know that `String` is a result of an interaction with the outside world. Similarly, if the type is `IO[Unit]`, you can be pretty sure that it wrote something to the outside world. (Note that I wrote those types using Scala syntax, not Haskell syntax.)

Second, when you're working on a large programming team, you know that a stressed-out programmer under duress can't accidentally slip an I/O function into a place where it shouldn't be.

> You know how it is: a deadline is coming up and the pressure is intense. Then one day someone on the programming team cracks and gives in to the pressure. Rather than doing something "the right way," he does something expedient, like accessing a database directly from a GUI method. "I'll fix it later," he rationalizes as he incurs Technical Debt. But as we know, *later never comes*, and the duct tape stays there until that day when you're getting ready to go on vacation and it all falls apart.

More … later

I'll explore this topic more in the I/O lessons in this book, but at this point I want to show that there is a *very different* way of thinking about I/O than what you might be used to in languages like C, C++, Java, C#, etc.

Summary

As I showed in this lesson, when you need to write I/O code in functional programming languages, the solution is to violate the "Only Write Pure Functions" rule. The general idea is that you write as much of your application as possible in an FP style, and then handle the UI, Database I/O, Web Service I/O, and File I/O in the best way possible for your current programming language and tools.

I also showed that wrapping your I/O functions in an `IO` type doesn't make a function pure, but it is a great way to add something to your function's type signature to let every know, "This function deals with I/O." When a function returns a type like `IO[String]` you can be very sure that it reached into the outside world to get that `String`, and when it returns `IO[Unit]`, you can be sure that it wrote something to the outside world.

What's next

So far I've covered a lot of background material about pure functions, and in the next lesson I share something that was an important discovery for me: The signatures of pure functions are much more meaningful than the signatures of impure functions.

See also

- The this Google Groups Scala debate where Martin Odersky states, "The `IO` monad does not make a function pure. It just makes it obvious that it's impure."

18

Pure Function Signatures Tell All

> *"In Haskell, a function's type declaration tells you a whole lot about the function, due to the very strong type system."*
>
> From the book,
> Learn You a Haskell for Great Good!

One thing you'll find in FP is that the signatures of pure functions tell you a lot about what those functions do. In fact, it turns out that the signatures of functions in FP applications are *much* more important than they are in OOP applications. As you'll see in this lesson:

> Because pure functions have no side effects, their outputs depend only on their inputs, and all FP values are immutable, pure function signatures tell you exactly what the function does.

OOP function signatures

When writing OOP applications I never gave much thought to method signatures. When working on development teams I always thought, "Meh, let me see the method source code so I can figure out what it *really* does." I remember one time a junior developer wrote what should have been a simple Java "setter" method named `setFoo`, and its source code looked something like this:

```
public void setFoo(int foo) {
    this.foo = foo;
    makeAMeal(foo);
```

```
    foo++;
    washTheDishes(foo);
    takeOutTheTrash();
}
```

In reality I don't remember everything that setter method did, but I clearly remember the `foo++` part, and then saw that it the `foo` and `foo++` values in other method calls. A method that —according to its signature — appeared to be a simple setter method was in fact much, much more than that.

I hope you can see the problem here: there's no way to know what's *really* happening inside an impure function without looking at its source code.

The first moral of this story is that because OOP methods *can* have side effects, I grew to only trust methods from certain people.

The second moral is that this situation can't happen with pure functions (at least not as blatanly as this).

Signatures of pure functions

The signatures of pure functions in Scala/FP have much more meaning than OOP functions because::

- They have no side effects
- Their output depends only on their inputs
- All values are immutable

To understand this, let's play a simple game.

A game called, "What can this pure function possible do?"

As an example of this — and as a first thought exercise — look at this function signature and ask yourself, "If `FOO` is a pure function, what can it possibly do?":

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

Ignore the name `FOO`; I gave the function a meaningless name so you'd focus only on the rest of the type signature to figure out what this function can possibly do.

To solve this problem, let's walk through some preliminary questions:

- Can this function read user input? It can't have side effects, so, no.
- Can it write output to a screen? It can't have side effects, so, no.
- Can it write (or read) information to (or from) a file, database, web service, or any other external data source? No, no, no, and no.

So what can it do?

If you said that there's an excellent chance that this function does one of the following things, pat yourself on the back:

- Converts a `String` to an `Int`
- Determines the length of the input string
- Calculates a hashcode or checksum for the string

Because of the rules of pure functions, those are the only types of things this function can do. Output depends only on input.

A second game example

Here's a second example that shows how the signatures of pure functions tell you a lot about what a function does. Given this simple class:

```
case class Person[name: String]
```

What can a pure function with this signature possibly do?:

```
def FOO(people: Seq[Person], n: Int): Person = ???
```

I'll pause to let you think about it …

By looking only at the function signature, you can guess that the function probably returns the nth element of the given `List[Person]`.

That's pretty cool. Because it's a pure function you know that the `Person` value that's returned must be coming from the `Seq[Person]` that was passed in.

Conversely, by removing the `n` parameter from the function:

```
def FOO(people: Seq[Person]): Person = ???
```

Can you guess what this function can do?

(Pause to let you think …)

My best guesses are:

- It's a `head` function
- It's a `tail` function
- It's a Frankenstein's Monster function that builds one `Person` from many `Persons`

A third game example

Here's a different variation of the "What can this pure function possibly do?" game. Imagine that you have the beginning of a function signature, where the input parameters are defined, but the return type is undefined:

```
def foo(s: String, i: Int) ...
```

Given only this information, can you answer the "What can this function possibly do?" question? That is, can you answer that question if you don't know what the function's return type is?

(Another pause to let you think …)

The answer is “no.” Even though `foo` is a pure function, you can’t tell what it does until you see its return type. But …

Even though you can’t tell *exactly* what it does, you can guess a little bit. For example, because output depends only on input, these return types are all allowed by the definition of a pure function:

```
def foo1(s: String, i: Int): Char = ???
def foo2(s: String, i: Int): String = ???
def foo3(s: String, i: Int): Int = ???
def foo4(s: String, i: Int): Seq[String] = ???
```

Even though you can’t tell what this function does without seeing its return type, I find this game fascinating. Where OOP method signatures had no meaning to me, I can make some really good guesses about what FP method signatures are trying to tell me — even when the function name is meaningless.

Trying to play the game with an impure method

Let’s look at one last example. What can this method possibly do?:

```
def foo(p: Person): Unit = ...
```

Because this method returns `Unit` (nothing), it can also be written this way:

```
def foo(p: Person) { ... }
```

In either case, what do you think this method can do?

Because it doesn’t return anything, it *must* have a side effect of some sort. You can’t know what those side effects are, but you can guess that it may do any or all of these things:

- Write to STDOUT
- Write to a file
- Write to a database

- Write to a web service
- Update some other variable(s) with the data in `p`
- Mutate the data in `p`
- Ignore `p` and do something totally unexpected

As you can see, trying to understand what an impure method can possibly do is much more complicated than trying to understand what a pure function can possibly do. As a result of this, I came to understand this phrase:

> Pure function signatures tell all.

Summary

As shown in this lesson, when a method has side effects there's no telling what it does, but when a function is pure its signature lets you make very strong guesses at what it does — even when you can't see the function name.

The features that make this possible are:

- The output of a pure function depends only on its inputs
- Pure functions have no side effects
- All values are immutable

What's next

Now that I've written several small lessons about pure functions, the next two lessons will show how combining pure functions into applications feels both like (a) algebra and (b) Unix pipelines.

19

Functional Programming as Algebra

> *"Some advanced Lispers will cringe when someone says that a function 'returns a value.' This is because Lisp derives from something called lambda calculus, which is a fundamental programming-like algebra developed by Alonzo Church. In the lambda calculus you 'run' a program by performing substitution rules on the starting program to determine the result of a function. Hence, the result of a set of functions just sort of magically appears by performing substitutions; never does a function consciously 'decide' to return a value. Because of this, Lisp purists prefer to say that a function 'evaluates to a result.'"*
>
> From the book, Land of Lisp

Introduction

I like to start most lessons with a relevant quote, but in the case of "FP as Algebra," several relevant quotes come to mind, so I'd like to share one more, from the book, Thinking Functionally with Haskell:

> "FP has a simple mathematical basis that supports equational reasoning about the properties of programs."

Because of functional programming's main features — pure functions and immutable values — writing FP code is like writing algebraic equations. Because I always liked algebra and thought it was simple, this made FP appealing to me.

I'll demonstrate what I mean in this lesson.

Goals

The first goal of this lesson is to give some examples of how FP code is like algebra.

A second goal of this lesson is to keep building an "FP way of thinking" about programming problems. The mindset of this lesson is that each pure function you write is like an algebraic equation, and then gluing those functions together to create a program is like combining a series of algebraic equations together to solve a math problem.

As this chapter's introductory quote states, when you begin to think about your functions as "evaluating to a result," you'll be in a state of mind where you're thinking about solving problems and writing your code as being like writing algebraic equations, and that's a good thing.

Background: Algebra as a reason for "Going FP"

Hopefully you'll find your own reasons for "Going FP," but for me the lightbulb went on over my head when I realized that FP let me look at my code this way. Gluing pure functions together felt like combining a series of algebraic equations together — i.e., algebraic substitution — and because I always liked algebra, this was a good thing.

Before learning FP my background was in OOP. I first learned and then taught Java and OOP in the 1990s and early 2000s, and with that background I always looked at problems from the eyes of an OOP developer. That never made me see writing code as being like writing mathematical expressions. I always thought, "Okay, these things here are my objects (`Pizza`, `Topping`, `Order`), these are their behaviors (`addTopping`), and they hide their internal workings from other objects."

But since learning FP I now see my code as being more like algebra, and it's a very different perspective. I clearly remember my first thought when I saw the connection between FP and algebra:

> "Whoa … if my function's output depends solely on its input, well, shoot, I can always write *one* pure function. If I can write one pure function, then I can write another, and then another. And then once they're all working I can glue them together to form a complete solution, like a series of equations. And since they're all pure functions they can't really fail — especially not because of hidden state issues — at least not if I test them properly."

Sometimes programming can get a little overwhelming when you think about writing an entire application, but when I realized that I can always write one pure function, that gave me a tremendous sense of confidence.

> As a programming tip, when you're writing a pure function, think of that function as your world, your only concern in the entire world. Because "output depends only on input," all you have to think about is that your function (your world) is given some inputs, and you need to create an algorithm to transform those inputs into the desired result.

Background: Defining algebra

It's important to understand what "algebra" is so you can really internalize this lesson.

Unfortunately, trying to find a good definition of algebra is difficult because many people go right from the concept of "algebra" to "mathematics," and that's not what I have in mind. This informal definition of algebra by Daniel Eklund fits my way of thinking a little better:

> For purposes of simplicity, let us define algebra to be two things: 1) a SET of objects (not "objects" as in object-oriented), and 2) the OPERATIONS used on those objects to create new objects from that set.

As emphasized, the key words in that sentence are *set* and *operations*. Mr. Eklund goes on to define "numeric algebra":

> In the case of *numeric* algebra — informally known as high-school algebra — the SET is the set of numbers (whether they be natural, rational, real, or complex) and the OPERATIONS used on these objects can be (but definitely not limited to be) addition or multiplication. The *algebra of numbers* is therefore the study of this set, and the laws by which these operators generate (or don't generate) new members from this set.

As an example, a *set* of natural numbers is [0,1,2 … infinity]. *Operations* on that set can be add, subtract, and multiply, and new members are generated using these operators, such as `1 + 2` yielding `3`.

Mr. Eklund goes on to define other types of algebras, but for our purposes I'll just share one more sentence:

> The key thing to realize here is that an algebra lets us talk about the objects and the operations abstractly, and to consider the laws that these operations obey as they operate on the underlying set.

In Scala/FP, the "objects" Mr. Eklund refers to can be thought of as the built-in Scala types and the custom types you create, and the "operations" can be thought of as the pure functions you write that work with those types.

For instance, in a pizza store application, the "set" might include types like `Pizza`, `Topping`, `Customer`, and `Order`. To find the operations that work with that set, you have to think about the problem domain. In a pizza store you add toppings to a pizza that a customer wants, and then you can add one or more pizzas to an order for that customer. The types are your set (the nouns), and the functions you create define the only possible operations (verbs) that can manipulate that set.

Given that discussion, a Scala `trait` for a `Pizza` type might look like this:

```
trait Pizza {
    def setCrustSize(s: CrustSize): Pizza
    def setCrustType(t: CrustType): Pizza
    def addTopping(t: Topping): Pizza
    def removeTopping(t: Topping): Pizza
    def getToppings(): Seq[Topping]
}
```

In the same way that `1` is a natural number and can work with operations like add and subtract, `Pizza` is a type and can work with the operations (methods) it defines.

From algebra to FP

If you haven't worked with algebra in a while, it may help to see a few algebraic functions as a refresher:

```
f(x) = x + 1
f(x,y) = x + y
f(a,b,c,x) = a * x^2 + b*x + c
```

It's easy to write those algebraic equations as pure functions in Scala/FP. Assuming that all the values are integers, they can be written as these functions in Scala:

```
def f(x: Int) = x + 1
def f(x: Int, y: Int) = x + y
def f(a: Int, b: Int, c: Int, x: Int) = a*x*x + b*x + c
```

These are pure functions ("output depends only on input") that use only immutable values. This shows one way that FP is like algebra by starting with algebraic functions and then writing the Scala/FP versions of those functions.

From FP to algebra

Similarly I can start with Scala/FP code and show how it looks like algebraic equations. For example, take a look at these Scala expressions:

```
val emailDoc = getEmailFromServer(src)
val emailAddr = getAddr(emailDoc)
val domainName = getDomainName(emailAddr)
```

You can see how that code is like algebra if I add comments to it:

```
val emailDoc = getEmailFromServer(src)     // val b = f(a)
val emailAddr = getAddr(emailDoc)          // val c = g(b)
val domainName = getDomainName(emailAddr)  // val d = h(c)
```

No matter what these functions do behind the scenes, they are essentially algebraic expressions, so you can reduce them just like you reduce mathematical expressions. Using simple substitution, the first two expressions can be combined to yield this:

```
val emailAddr = getAddr(getEmailFromServer(src))
val domainName = getDomainName(emailAddr)
```

Then those two expressions can be reduced to this:

```
val domainName = getDomainName(getAddr(getEmailFromServer(src)))
```

If you look at the comments I added to the code, you'll see that I started with this:

```
val b = f(a)
val c = g(b)
val d = h(c)
```

and reduced it to this:

```
val d = h(g(f(a)))
```

I can make these substitutions because the code is written as a series of expressions that use pure functions.

You can write the code in the three lines, or perform the substitutions to end up with just one line. Either approach is valid, and equal.

What makes this possible is that other than `getEmailFromServer(src)`, which is presumably an impure function, the code:

- Only uses pure functions (no side effects)
- Only uses immutable values

When your code is written like that, it really is just a series of algebraic equations.

Benefit: Algebra is predictable

A great thing about algebra is that the results of algebraic equations are incredibly predictable. For example, if you have a `double` function like this:

```
def double(i: Int) = i * 2
```

you can then call it with the number `1` an infinite number of times and it will always return `2`. That may seem obvious, but hey, it's how algebra works.

Because of this, you know that these things will *always* happen:

```
println(double(1))   // prints 2
println(double(2))   //   "    4
println(double(3))   //   "    6
```

And you also know that this can *never* happen:

```
println(double(1)) // prints 5  (can never happen)
println(double(1)) // prints 17 (can never happen)
```

With pure functions you can never have two different return values for the same input value(s). This can't happen with pure functions, and it can't happen with algebra, either.

A game: What can possibly go wrong?

A great thing about thinking about your code as algebra is that you can look at one of your pure functions and ask, "What can possibly go wrong with this function?" When you do so, I hope that tring to find any problems with it will be very difficult. After thinking about it long and hard I hope you get to the point of saying, "Well, I guess the JVM could run out of RAM (but that doesn't have anything directly to do with my function)."

My point is that because it's isolated from the rest of the world, it should be a real struggle to think about how your pure function can possibly fail. When you're writing OOP code you have to concern yourself that "output *does not* only depend on input," which means that you have to think about everything else in the application that can fail or be a problem — i.e., things like (a) state of the application outside the function's scope, and (b) variables being mutated while you're trying to use them — but with FP code you don't have those concerns.

For example, imagine that you're writing a multi-threaded imperative application, you've been given a list of users, and the purpose of your function is to sort that list of users. There are a lot of ways to sort lists, so that isn't hard, but what happens to your code if that list of users is mutated by another thread while your function is trying to sort the list? For instance, imagine that 20 users are removed from the list while you're trying to sort it; what will happen to your function?

You can demonstrate this problem for yourself. Remembering that Scala `Array` elements can be mutated, imagine that you have an `Array[String]` like this:

```
// 1 - a mutable sequence to work with
val arr = Array("one", "two", "three", "four", "five")
```

Then imagine that you begin printing the length of each string in a different thread, like this:

```
// 2 - start printing the numbers in a different thread
val thread = new Thread {
    override def run {
        printStringLength(arr)
    }
}
thread.start
```

If you now mutate the array like this:

```
// 3 - mutate the sequence to see how that other thread works
Thread.sleep(100)
arr(3) = null
```

you can easily generate a `NullPointerException` if your `printStringLength` method looks like this:

```
def printStringLength(xs: Seq[String]) {
    for (x <- xs) {
        println(x.length)
        Thread.sleep(200)
    }
}
```

Conversely, it's impossible to replicate this example if you use a Scala `Vector` or `List`. Because these sequences are immutable, you can't accidentally mutate a sequence in one thread while it's being used in another.

Transform as you copy, don't mutate

In my previous Java/OOP life I mutated variables all the time. That's how I did almost everything, and frankly, I didn't know there was another way. I knew that a Java `String` was immutable, but based on my OOP thinking, I thought this was more of a pain than anything that was actually helpful to me.

But when you think of your code as algebra, you realize that mutating a variable has nothing to do with algebra. For instance, I never had a math instructor who said, "Okay, `x` is currently 10, but let's go ahead and add 1 to it so `x` is now 11." Instead what they said is, "Okay, we have `x`, which is 10, and what we'll do is add 1 to it to get a new value `y`":

```
x = 10
y = x + 1
```

In FP code you do the same thing. You never mutate `x`, but instead you use it as a foundation to create a new value. In Scala, you typically do this using the case class `copy` method.

Case class `copy` *method*

When you use a Scala *case class* you automatically get a `copy` method that supports this "transform as you copy" algebraic philosophy.

A simple way to demonstrate this is to show what happens when a person changes their name. I'll demonstrate this with two variations of a `Person` class, first showing an OOP/imperative approach, and then showing an FP/algebraic approach.

With OOP code, when Jonathan Stuart Leibowitz changes his name to Jon Stewart, you might write code like this:

```
// oop design
class Person(var name: String)

// create an instance with the original name
var p = new Person("Jonathan Stuart Leibowitz")

// change the name by mutating the instance
p.name = "Jon Stewart"
```

In my OOP life I wrote code like that all the time and never gave it a second thought. But you just don't do that sort of thing in algebra. Instead, what you do in FP/algebraic code is this:

```
// fp design
case class Person(name: String)

// create an instance with the original name
val p = Person("Jonathan Stuart Leibowitz")

// create a new instance with the "transform as you copy" approach
val p2 = p.copy(name = "Jon Stewart")
```

The FP approach uses the `copy` method to create a new value `p2` from the original `p`, resulting in `p2.name` being "Jon Stewart."

Mathematically, the last two lines of the FP approach are similar to this:

```
val x = a
val y = x + b
```

Or, if it helps to use the original value names, this:

```
val p  = a
val p2 = p + b
```

It's good to see the case class `copy` approach now, because (a) it's a Scala/FP idiom, and (b) we're going to use it a lot in this book.

> As I mentioned earlier, I *never* thought of my OOP code as having the slightest thing to do with algebra. Now I think of it that way all the time, and that thought process is the result of writing pure functions and using only immutable variables.

Benefit: Automated property-based testing

In a preview of a later chapter, a nice benefit of coding in this style is that you can take advantage of something called "property-based testing," what I'll call "PBT" here. PBT is a way of testing your code in a manner similar to using JUnit, but instead of writing each individual test manually at a low level, you instead *describe* your function and let the PBT testing tool pound away at it. You can tell the PBT tool to throw 100 test values at your function, or 1,000, or many more, and because your function is a pure function — and therefore has this algebraic property to it — the PBT library can run tests for you.

Technically you *can* probably do the same thing with impure functions, but I find that this technique is much easier with pure functions.

> I wrote a little about this in the Benefits of Functional Programming lesson, and I write much more about it later in this book, so I won't write any more here. If you're interested in more details at this time, see the ScalaCheck website and the property-based testing page on that site.

Later in this book: Algebraic Data Types

Another way that FP relates to algebra is with a concept known as Algebraic Data Types, or ADTs. Don't worry about that name, ADT is a simple concept. For example, this code is an ADT:

```
sealed trait Bool
case object True extends Bool
case object False extends Bool
```

This code from the book, Beginning Scala, is also an ADT:

```
sealed trait Shape
case class Circle(radius: Double) extends Shape
case class Square(length: Double) extends Shape
case class Rectangle(h: Double, w: Double) extends Shape
```

I don't want to get into this in much detail right now, I just wanted to let you know that there's more algebra later in this book. The "algebra" in ADTs is described on the Haskell wiki like this:

> "Algebraic" refers to the property that an Algebraic Data Type is created by "algebraic" operations. The "algebra" here is "sums" and "products" (of types).

Again, don't fear the term; it's another complicated-sounding term for a simple concept, as shown in these examples.

Summary

In this lesson I tried to show a few ways that functional programming is like algebra. I showed how simple algebraic functions can be written as pure functions in Scala, and I showed how a series of Scala expressions looks just like a series of algebraic functions. I also demonstrated how a series of expressions can be reduced using simple algebraic substitution. I also noted that in the future you'll learn about a term named Algebraic Data Types.

The intent of this lesson is to help you keep building an "FP way of thinking" about programming problems. If you write your code using only pure functions and immutable variables, your code will natural migrate towards this algebraic way of thinking:

> Pure Functions + Immutable Values == Algebra

Who knows, you may even start saying that your functions "evaluate to a result."

What's next

In the next chapter I'll make a quick observation that when you write functional code, you're also writing code that fits a style known as Expression-Oriented Programming.

See Also

- What the Heck are Algebraic Data Types, the Daniel Eklund paper
- Algebraic Data Type on Wikipedia
- The Algebra of Algebraic Data Types
- Algebraic Data Type on the Haskell wiki

20

A Note About Expression-Oriented Programming

> *"Statements do not return results and are executed solely for their side effects, while expressions always return a result and often do not have side effects at all."*
>
> From the Wikipedia page on Expression-Oriented Programming

Goals

This chapter isn't a *lesson* so much as it as an observation — a short note that the FP code I'm writing in this book also falls into a category known as *Expression-Oriented Programming*, or EOP.

In fact, because Pure FP code is more strict than EOP, FP is a superset of EOP. As a result, we just happen to be writing EOP code while we're writing Scala/FP code.

Therefore, my goals for this lesson are:

- To show the difference between *statements* and *expressions*
- To briefly explain and demonstrate EOP
- To note that all "Pure FP" code is also EOP code

I wrote about EOP in the Scala Cookbook, so I'll keep this discussion short.

Statements and expressions

When you write pure functional code, you write a series of expressions that combine pure functions. In addition to this code conforming to an FP style, the style also fits the definition of "Expression-Oriented Programming," or EOP. This means that every line of code returns a result ("evaluates to a result"), and is therefore an *expression* rather than a *statement*.

> As noted in the quote at the beginning of this chapter, statements do not return results and are executed solely for their side effects.

An expression has the form:

```
val resultingValue = somePureFunction(someImmutableValues)
```

Contrast that with the OOP "statement-oriented code" I used to write:

```
order.calculateTaxes()
order.updatePrices()
```

Those two lines of code are *statements* because they don't have a return value; they're just executed for their side effects.

In FP and EOP you write those same statements as *expressions*, like this:

```
val tax = calculateTax(order)
val price = calculatePrice(order)
```

While that may seem like a minor change, the effect on your overall coding style is huge. Writing code in an EOP style is essentially a gateway to writing in an FP style.

> I'm tempted to write about "The Benefits of EOP," but because I already wrote about "The Benefits of Functional Programming" in a previous lesson, I won't repeat those points here. Please see those chapters to refresh your memory on all of those benefits.

A key point

A key point of this lesson is that when you see statements like this:

```
order.calculateTaxes()
order.updatePrices()
```

you should think, “Ah, these are *statements* that are called for their side effects. This is imperative code, not FP code.”

Scala supports EOP (and FP)

As I noted in the Scala Cookbook, these are obviously *expressions*:

```
val x = 2 + 2
val doubles = List(1,2,3,4,5).map(_ * 2)
```

But it’s a little less obvious that the if/then construct can also be used to write expressions:

```
val greater = if (a > b) a else b
```

> Note: In Java you need the special ternary operator syntax to write code like that.

The `match` construct also returns a result, and is used to write expressions:

```
val evenOrOdd = i match {
    case 1 | 3 | 5 | 7 | 9 => println("odd")
    case 2 | 4 | 6 | 8 | 10 => println("even")
}
```

And try/catch blocks are also used to write expressions:

```
def toInt(s: String): Int = {
    try {
        s.toInt
    } catch {
        case _ : Throwable => 0
    }
}
```

> As you'll see in the upcoming lessons on recursion, `match` expressions are a big part of the Scala language, and because they evaluate to a value, you'll often write the first part of recursive functions like this:
>
> ```
> def sum(list: List[Int]): Int = list match { ...
> ```

Summary

When every line of code has a return value it is said that you are writing *expressions*, and using an EOP style. In contrast, *statements* are lines of code that do not have return values, and are executed for their side effects. When see statements in code you should think, "This is imperative code, not FP code."

As noted in this lesson, because EOP is a subset of an FP style, when you write Scala/FP code you are also writing EOP code.

What's next

Given this background, the next lesson shows how writing Unix pipeline commands also fits an EOP style, and in fact, an FP style.

21

Functional Programming is Like Unix Pipelines

> *"Pipes facilitated function composition on the command line. You could take an input, perform some transformation on it, and then pipe the output into another program. This provided a very powerful way of quickly creating new functionality with simple composition of programs. People started thinking how to solve problems along these lines."*
>
> Alfred Aho, one of the creators of the AWK programming language, in the book, Masterminds of Programming

Goals

The primary goal of this lesson is to show that you can think of writing functional programs as being like writing Unix pipeline commands. Stated another way, if you've written Unix pipeline commands before, you have probably written code in a functional style, whether you knew it or not.

As a second, smaller goal, I want to demonstrate a few ways that you can look at your code visually to help you "Think in FP."

> Note: This section is written for Unix and Linux users. If you don't know Unix, (a) I highly recommend learning it, and (b) you may want to (sadly) skip this section, as it may not make much sense unless you know the Unix commands that I show.

Discussion

One way to think about FP is that it's like writing Unix/Linux pipeline commands, i.e., a series of two or more commands that you combine at the Unix command line to get a desired result.

For example, imagine that your boss comes to you and says, "I need a script that will tell me how many unique users are logged into a Unix system at any given time." How would you solve this problem?

Knowing Unix, you know that the `who` command shows the users that are currently logged in. So you know that you want to start with `who` — that's your data source. To make things interesting, let's assume that `who` doesn't support any command-line arguments, so all you can do is run `who` without arguments to generate a list of users logged into your system, like this:

```
$ who
al       console  Oct 10 10:01
joe      ttys000  Oct 10 10:44
tallman  ttys001  Oct 10 11:05
joe      ttys002  Oct 10 11:47
```

`who`'s output is well structured and consistent. It shows the username in the first column, the "console" they're logged in on in the second column, and the date and time they logged in on in the last columns.

> Some Unix systems may show the IP address the user is logged in from.
> I left that column off of these examples to keep things simple.

If you didn't have to automate this solution, you could solve the problem by looking at the unique usernames in the first column. In this case there are four lines of output, but only three of the usernames are unique — `al`, `joe`, and `tallman` — so the current answer to your boss's question is that there are three unique users logged into the system at the moment.

Now that you know how to solve the problem manually, the question becomes, how do you automate this solution?

An algorithm

The solution's algorithm appears to be:

- Run the `who` command
- Create a list of usernames from the first column
- Get only the unique usernames from that list
- Count the size of that list

In Unix that algorithm translates to chaining these commands together:

- Start with `who` as the data source
- Use a command like `cut` to create the list of usernames
- Use `uniq` to get only the unique usernames from that list
- Use `wc -l` to count those unique usernames

Implementing the algorithm

A good solution for the first two steps is to create this simple Unix pipeline:

```
who | cut -d' ' -f1
```

That `cut` command can be read as, "Using a blank space as the field separator (`-d' '`), print the first field (`-f1`) of every row of the data stream from STDIN to STDOUT." That pipeline command results in this output:

```
al
joe
tallman
joe
```

Notice what I did here: I combined two Unix commands to get a desired result. If you think of the `who` command as providing a list of strings, you can think of the `cut` command as being a pure function: it takes a list of strings as an input parameter, runs a transformation algorithm on that incoming data, and produces an output list of strings. It doesn't use anything but the incoming data and its algorithm to produce its result.

As a quick aside, the signature for a Scala `cut` function that works like the Unix `cut` command might be written like this:

```
def cut(strings: Seq[String],
        delimiter: String,
        field: Int): Seq[String] = ???
```

Getting back to the problem at hand, my current pipeline command generates this output:

```
al
joe
tallman
joe
```

and I need to transform that data into a "number of unique users."

To finish solving the problem, all I need to do is to keep combining more pure functions — er, Unix commands — to get the desired answer. That is, I need to keep transforming the data to get it into the format I want.

The next thing I need to do is reduce that list of *all users* down to a list of *unique users*. I do that by adding the `uniq` command to the end of the current pipeline:

```
who | cut -d' ' -f1 | uniq
```

`uniq` transforms its STDIN to this STDOUT:

```
al
joe
tallman
```

Now all I have to do to get the number of unique users is count the number of lines that are in the stream with `wc -l`:

```
who | cut -d' ' -f1 | uniq | wc -l
```

That produces this output:

```
       3
```

Whoops. What's that `3` doing way over there to the right? I want to think of my result as being an `Int` value, but this is more like a `String` with a bunch of leading spaces. What to do?

Well, it's Unix, so all I have to do is add another command to the pipeline to transform this string-ish result to something that works more like an integer.

There are many ways to handle this, but I know that the Unix `tr` command is a nice way to remove blank spaces, so I add it to the end of the current pipeline:

```
who | cut -d' ' -f1 | uniq | wc -l | tr -d ' '
```

That gives me the final, desired answer:

```
3
```

That looks more like an integer, and it won't cause any problem if I want to use this result as an input to some other command that expects an integer value (with no leading blank spaces).

> If you've never used the `tr` command before, it stands for *translate*, and I wrote a few `tr` command examples many years ago.

The solution as a shell script

Now that I have a solution as a Unix pipeline, I can convert it into a little shell script. For the purposes of this lesson, I'll write it in a verbose manner rather than as a pipeline:

```
WHO=`who`
RES1=`echo $WHO  | cut -d' ' -f1`
```

```
RES2=`echo $RES1 | uniq`
RES3=`echo $RES2 | wc -l`
RES4=`echo $RES3 | tr -d ' '`
echo $RES4
```

Hmm, that looks suspiciously like a series of *expressions*, followed by a print statement, doesn't it? Some equivalent Scala code might look like this:

```
val who: Seq[String] = getUsers   // an impure function
val res1 = cut(who, " ", 1)
val res2 = uniq(res1)
val res3 = countLines(res2)
val res4 = trim(res3)
println(res4)                     // a statement
```

Combining simple expressions

I usually write "one expression at a time" code like this when I first start solving a problem, and eventually see that I can combine the expressions. For example, because the first and last lines of code are impure functions I might want to leave them alone, but what about these remaining lines:

```
val res1 = cut(who, " ", 1)
val res2 = uniq(res1)
val res3 = countLines(res2)
val res4 = trim(res3)
```

In the first line, because `cut` is a pure function, `res1` and `cut(who, " ", 1)` will always be equivalent, so I can eliminate `res1` as an intermediate value:

```
val res2 = uniq(cut(who, " ", 1))
val res3 = countLines(res2)
val res4 = trim(res3)
```

Next, because `res2` is always equivalent to the right side of its expression, I can eliminate `res2` as an intermediate value:

```
val res3 = countLines(uniq(cut(who, " ", 1)))
val res4 = trim(res3)
```

Then I eliminate `res3` for the same reason:

```
val res4 = trim(countLines(uniq(cut(who, " ", 1))))
```

Because there are no more intermediate values, it makes sense to rename `res4`:

```
val result = trim(countLines(uniq(cut(who, " ", 1))))
```

If you want, you can write the entire original series of expressions and statements — including `getUsers` and the `println` statement — like this:

```
println(trim(countLines(uniq(cut(getUsers, " ", 1)))))
```

As a recap, I started with this:

```
val who: Seq[String] = getUsers
val res1 = cut(who, " ", 1)
val res2 = uniq(res1)
val res3 = countLines(res2)
val res4 = trim(res3)
println(res4)
```

and ended up with this:

```
println(trim(countLines(uniq(cut(getUsers, " ", 1)))))
```

The thing that enables this transformation is that all of those expressions in the middle of the original code are pure function calls.

This is the Scala equivalent of the Unix pipeline solution:

```
who | cut -d' ' -f1 | uniq | wc -l | tr -d ' '
```

> I always find solutions like this amusing, because if you have ever seen Lisp code, condensed Scala/FP code tends to look like this, where you read the solution starting at the inside (with `getUsers`), and work your way out (to `cut`, then `uniq`, etc.).

Note: You don't have to use this condensed style. Use whatever you're comfortable with.

How is this like functional programming?

"That's great," you say, "but how is this like functional programming?"

Well, if you think of the `who` command as generating a list of strings (`Seq[String]`), you can then think of `cut`, `uniq`, `wc`, and `tr` as being a series of transformer functions, because they transform the input they're given into a different type of output, as shown in Figure 21.1.

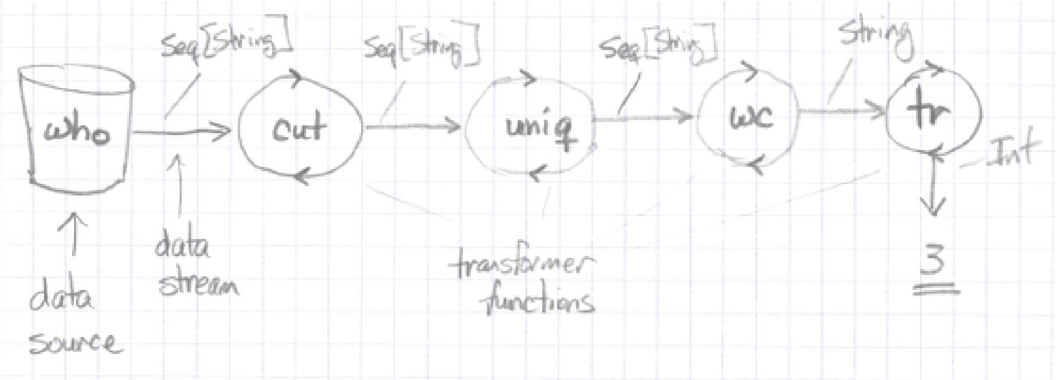


Figure 21.1: Unix commands transform their input into their output.

Looking at just the `wc` command — and thinking of it as a pure function — you can think of it as taking a `Seq[String]` as its first input parameter, and when it's given the `-l` argument, it returns the number of lines that it counts in that `Seq`.

In these ways the `wc` command is a pure function:

- It takes a `Seq[String]` as input

- It does not rely on any other state or hidden values
- It does not read or write to any files
- It does not alter the state of anything else in the universe
- Its output depends only on its input
- Given the same input at various points in time, it always returns the same value

The one thing that `wc` did that I didn't like is that it left-pads its output with blank spaces, so I used the `tr` command just like the `wc` command to fix that problem: as a pure function.

A nice way to think of this code is like this:

```
Input -> Transformer -> Transformer ... Transformer-> Output
```

With that thought, this example looks as shown in Figure 21.2.

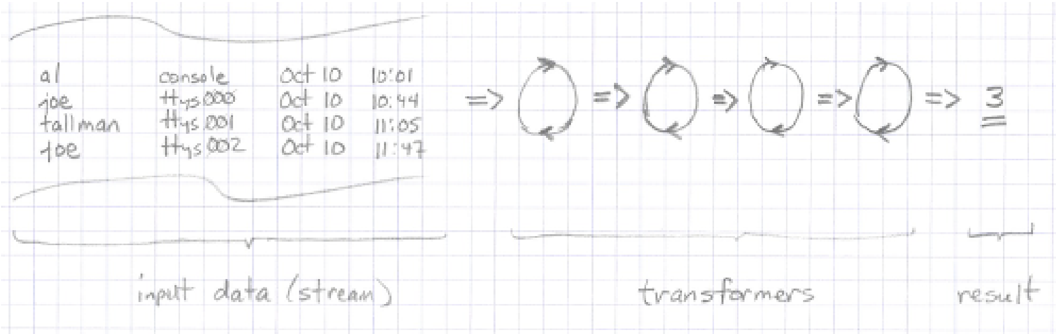


Figure 21.2: Using a series of transformers in a pipeline to solve a problem.

Note a few key properties in all of this. First, data flows in only one direction, as shown in Figure 21.3.

Second, Figure 21.4 shows that the input data a function is given is never modified.

Finally, as shown in Figure 21.5, you can think of functions as having an *entrance* and an *exit*, but there are no side doors or windows for data to slip in or out.

These are all important properties of pure functions (and Unix commands).

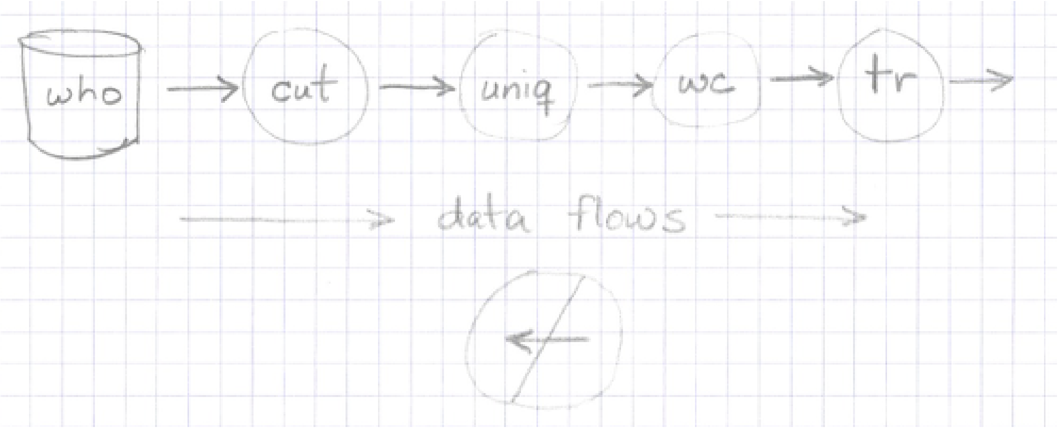


Figure 21.3: Pipeline data flows in only one direction.

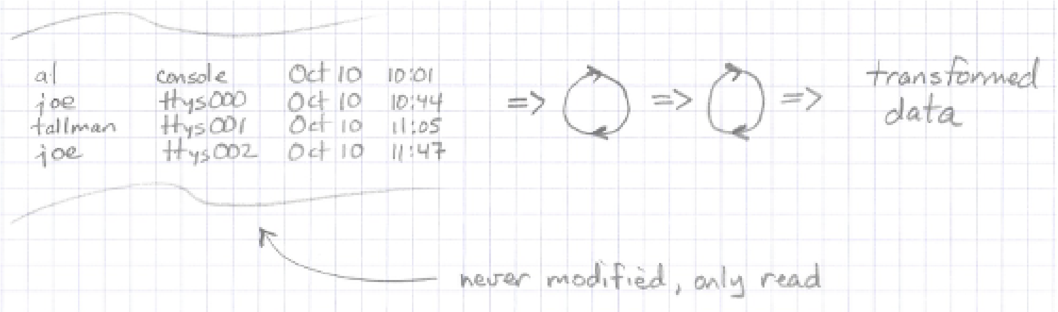


Figure 21.4: Data is never modified.

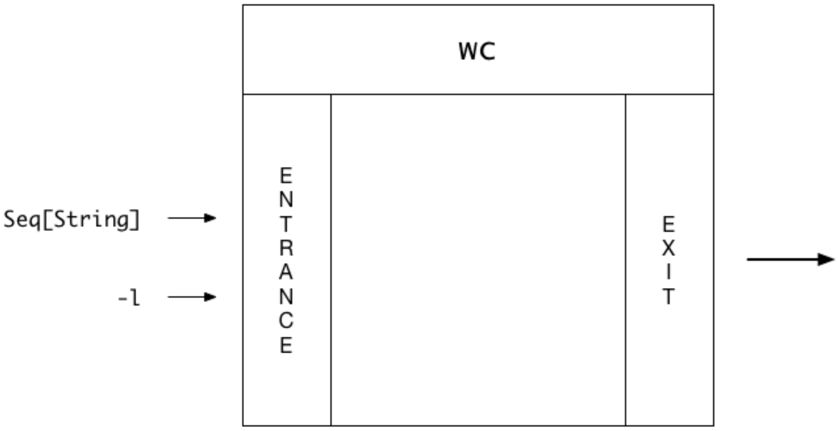


Figure 21.5: Pure functions have one entrance and one exit.

Pipelines as combinators

There's another interesting point about this example in regards to FP. When I combine these commands together like this:

```
who | cut -d' ' -f1 | uniq | wc -l | tr -d ' '
```

I create what's called in Unix a *pipeline* or *command pipeline*. In FP we call that same thing a *combinator*. That is, I combined the three commands — pure functions — together to get the data I wanted.

If I had structured my Scala code differently I could have made it look like this:

```
who.cut(delimiter=" ", field=1)
   .uniq
   .wc(lines = true)
   .tr(find=" ", replace="")
```

I'll add a more formal definition of "combinator" later in this book, but in general, when you see code like this — a chain of functions applied to some initial data — this is what most people think when they use the term "combinator." This is another case where an FP term sounds scary, but remember that whenever you hear the term "combinator" you can think "Unix pipeline."

Look back at how you thought about that problem

At this point it's worth taking a moment to think about the thought process involved in solving this problem. If you look back at how it was solved, our thinking followed these steps:

- We started with the problem statement: wanting to know how many users are logged into the system.
- We thought about what data source had the information we needed, in this case the output of the `who` command.
- At this point should note that implicit in my own thinking is that I knew the *structure* of the data I'd get from the `who` command. That is, as an experienced

Unix user I knew that `who` returns a list of users, with each user login session printed on a new line.

- Depending on your thought process you may have thought of the `who` output as a multiline `String` or as a `List` (or more generally as a `Seq` in Scala). Either thought is fine.
- Because you knew the structure of the `who` data, and you know your Unix commands, you knew that you could apply a sequence of standard commands to the `who` data to get the number of unique users.
- You may or may not have known beforehand that the `wc -l` output is padded with blank spaces. I did not.

The functional programming thought process

The reason I mention this thought process is because that's what the functional programming thought process is like:

- You start with a problem to solve.
- You either know where the data source is, or you figure it out.
- Likewise, the data is either in a known format, or in a format you need to learn.
- You clearly define the output you want in the problem statement.
- You apply a series of pure functions to the input data source(s) to transform the data into a new structure.
- If all of the functions that you need already exist, you use them; otherwise you write new pure functions to transform the data as needed.

> Note the use of the word *apply* in this discussion. Functional programmers like to say that they *apply* functions to input data to get a desired output. As you saw, using the word "apply" in the previous discussion was quite natural.

A second example

As a second example of both "Unix pipelines as FP," and "The FP thought process," imagine that you want a sorted list of all users who are currently logged in. How would you get the desired answer?

Let's follow that thought process again. I'll give you the problem statement, and everything after that is up to you.

a) You start with a problem to solve.

Problem statement: I want a sorted list of all users who are currently logged in.

b) You either know where the data source is, or you figure it out.

The data source is:

c) Likewise, the data is either in a known format, or in a format you need to learn.

The data format looks like this:

d) Going back to the problem statement, you clearly define the output you want

The desired output format is:

e) Apply a series of functions to the input data source(s) to get the output data you want.

The command pipeline needed to get the output data from the input data is:

f) If all of the functions that you need already exist, you use them; otherwise you write new pure functions to convert/transform the data as needed.

Do you need to create any new functions to solve this problem? If so, define them here:

One possible solution

Here's my solution:

```
who | cut -f1 -d' ' | uniq | sort
```

More exercises

That exercise was intentionally a relatively simple variation of the original exercise. Here are a few more advanced exercises you can work to get the hang of this sort of problem solving:

- Write a pipeline to show the number of processes owned by the root user.
- Write a pipeline to show the number of open network connections. (Tip: I use `netstat` as the data source.)

- Use the `lsof` command to show what computers your computer is currently connected to.
- Write a pipeline command to show which processes are consuming the most RAM on your computer.
- Write a command to find the most recent *.gitignore* file on your computer.

Data flow diagrams

Besides demonstrating how writing Unix pipeline commands are like writing FP code (and vice-versa), I'm also trying to demonstrate "The FP Thought Process." Because "output depends only on input," FP lends itself to something that used to be called "Data Flow Diagrams" — or DFDs — back in the old days.

There's a formal notation for DFDs, but I don't care for it. (There are actually several formal notations.) If I was going to sketch out the solution to the last problem, I'd draw it like the image in Figure 21.6.

Because I'm using my own graphical drawing language here, I'll note that at the moment:

- I prefer to draw *data flows* as streams (simple tables).
- I like to annotate streams with their data types.
- I like to draw functions as rectangles (because of the whole front-door/back-door, entrance/exit concept).

I'm not suggesting that you have to draw out every problem and solution like this, but if you're working on a hard problem, this can be helpful.

"Conservation of data"

If I'm working on a difficutl problem, or trying to explain a solution to other people, I like to draw visual diagrams like that. The book, Complete Systems Analysis, by

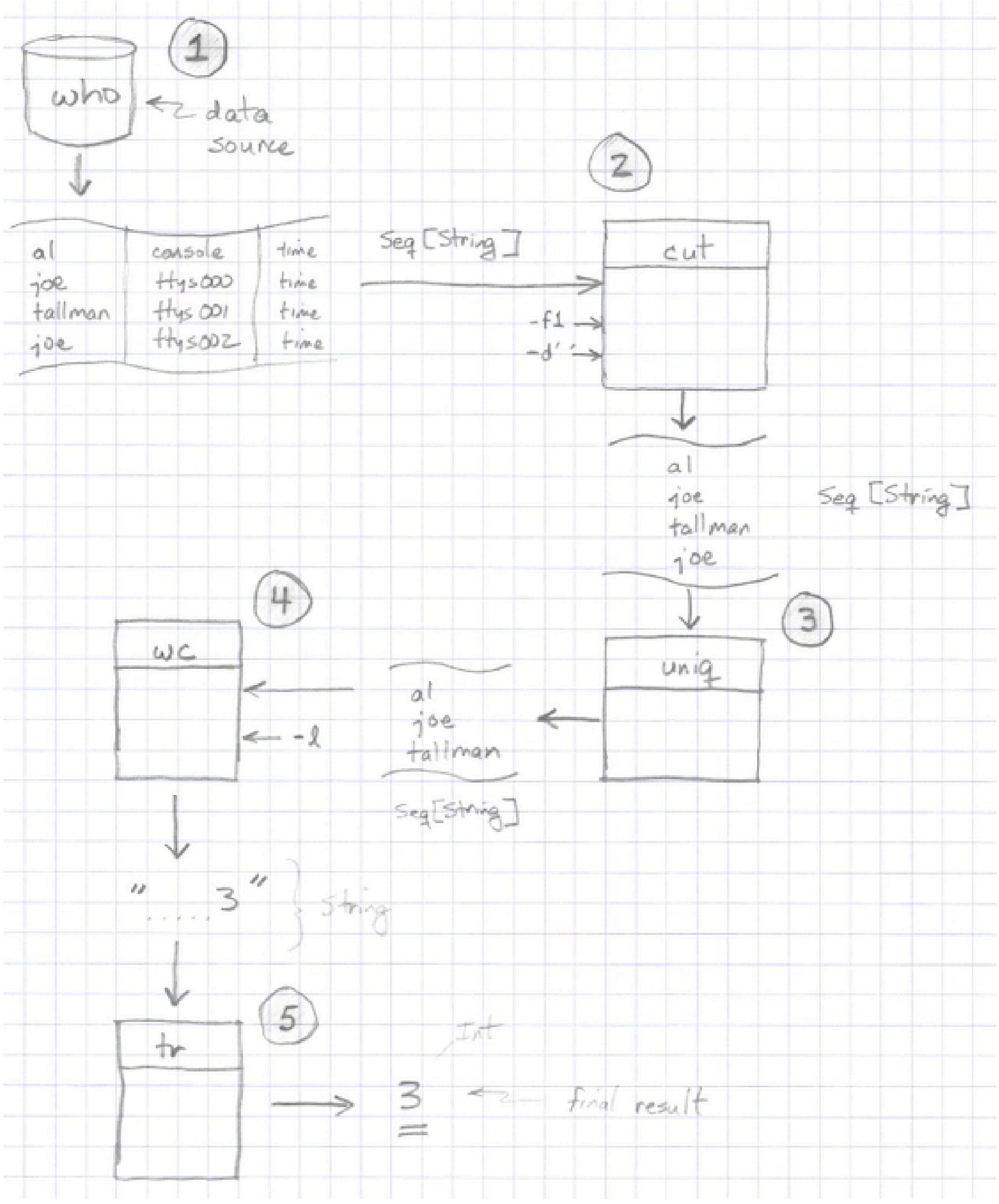


Figure 21.6: A DFD-like sketch of the pipeline solution.

Robertson and Robertson, defines something else that they call a “Rule of Data Conservation,” which they state like this:

> “Each process (function) in the data flow diagram must be able to produce the output *data flows* from its input.”

Using their diagramming process, the data that flows from the `who` command would be described like this:

```
Who = Username + Terminal + Date + Time
```

If you take the time to draw the data flows like this, it’s possible to make sure that the “Rule of Data Conservation” is satisfied — at least assuming that you know each function’s algorithm.

“Black holes and miracles”

A set of Power Point slides at DePaul.edu (that is hard to link to because of the whole “PPT” thing) makes the following observations about data flows:

- Data stays at rest unless moved by a process
- Processes cannot consume or create data
    - Must have at least 1 input data flow (to avoid miracles)
    - Must have at least 1 output data flow (to avoid black holes)

Just substitute “function” for “process” in their statements, and I really like those last two lines — avoiding *black holes* and *miracles* — as they apply to writing pure functions.

One caveat about this lesson

In this lesson I tried to show how writing Unix pipeline commands is like writing FP code. This is true in that combining Unix commands to solve a problem is like combining pure functions to solve a problem.

One part I didn't show is a program that runs continuously until the user selects a "Quit" option. But fear not, I will show this in an upcoming lesson, I just need to provide a little more background information, including covering topics like recursive programming.

Summary

As I mentioned at the beginning, my main goal for this lesson is to demonstrate that writing Unix pipeline commands is like writing functional code. Just like functional programming, when you write Unix pipeline commands:

- You have data sources, or inputs, that bring external data into your application.
- Unix commands such as `cut`, `uniq`, etc., are like pure functions. They take in immutable inputs, and generate output based only on those inputs and their algorithms.
- You combine Unix commands with pipelines in the same way that you use FP functions as "combinators."

See Also

- `tr` command examples on my website
- Unix pipelines on Wikipedia
- Data Flow Diagrams on Wikipedia
- Data Flow Diagrams on visual-paradigm.com

22

Functions Are Variables, Too

> *"A variable is a named entity that refers to an object. A variable is either a val or a var. Both vals and vars must be initialized when defined, but only vars can be later reassigned to refer to a different object."*
>
> The Scala Glossary

Goals

The goal of this lesson is to show that in a good FP language like Scala, you can use functions as values. In the same way that you create and use `String` and `Int` values, you can use a function:

```
val name = "Al"                    // string value
val weight = 222                   // int value
val double = (i: Int) => i * 2     // function value
```

To support this goal, this lesson shows:

- How to define a function as a `val`
- The "implicit" form of the `val` function syntax
- How to pass a function to another function
- Other ways to treat functions as values

Scala's `val` function syntax

Understanding Scala's `val` function syntax is important because you'll see function signatures over and over in a variety of places, including:

- When you define `val` functions
- When you define function input parameters (i.e., when one function takes another function as an input parameter)
- When you're reading the Scaladoc for almost every method in the Scala collections classes
- In the REPL output

You'll see examples of most of these in this lesson.

Formalizing some definitions

Before getting into this lesson, it will help to make sure that I'm formal about how I use certain terminology. For instance, given this expression:

```
val x = 42
```

it's important to be clear about these things:

1) Technically, `x` is a *variable*, a specific type of variable known as an *immutable variable*. Informally, I prefer to refer to `x` as a "value," as in saying, "`x` is an integer value." I prefer this because `x` is declared as a `val` field; it's bound to the `Int` value 42, and that can never change. But to be consistent with (a) other programming resources as well as (b) algebraic terminology, I'll refer to `x` as a *variable* in this lesson.

> Wikipedia states that in algebra, "a variable is an alphabetic character representing a number, called the value of the variable, which is either arbitrary or not fully specified or unknown." So in this way, referring to `x` as a variable is consistent with algebraic terms.

2) `x` has a *type*. In this case the type isn't shown explicitly, but we know that the type is an `Int`. I could have also defined it like this:

   ```
   val x: Int = 42
   ```

But because programmers and the Scala compiler know that `42` is an `Int`, it's convenient to use the shorter form.

3) Variables themselves have *values*, and in this example the variable `x` has the value `42`. (As you can imagine, it might be confusing if I wrote, "The value `x` has the value `42`.)"

I'm formalizing these definitions now because as you're about to see, these terms also apply to creating functions: functions also have variable names, types, and values.

Function literals

If you haven't heard of the term "function literal" before, it's important to know that in this example:

```
xs.map(x => x * 2)
```

this part of the code is a *function literal*:

```
x => x * 2
```

It's just like saying that this is a *string literal*:

```
"hello, world"
```

I mention this because …

Function literals can be assigned to variables

In functional programming languages, function literals can be assigned to variable names. In Scala this means:

- You can define a function literal and assign it to a `val` field, which creates an immutable variable
- You give that variable a name, just like any other `val` field
- A function variable has a value, which is the code in its function body
- A function variable has a *type* — more on this shortly
- You can pass a function around to other functions, just like any other `val`
- You can store a function in a collection, such as a `Map`
- In general, you use a function variable just like any other variable

The `val` function syntax

In the "Explaining the `val` Function Syntax" appendix, I show two different ways to define functions using `val`s in Scala. In this lesson I'll use only the following approach, which shows the "*implicit* return type" syntax:

```
val isEven = (i: Int) => i % 2 == 0
```

In this case "implicit" means that this function doesn't *explicitly* state that it returns a `Boolean` value; both you and the compiler can infer that by looking at the function body.

> Scala also has a `val` function syntax where you *explicitly* declare the function's return type, and I show that in the appendix.

I discuss the implicit syntax in detail in the appendix, but Figure 22.1 shows a quick look at what each of those fields means.

If that syntax looks a little unusual, fear not, I show more examples of it in this lesson and in the appendices.

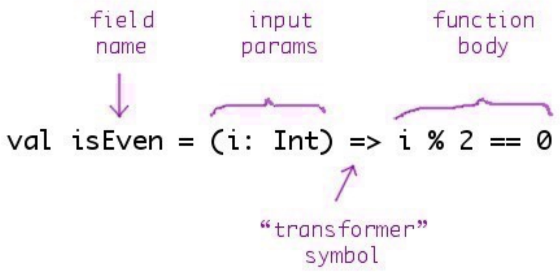


Figure 22.1: Scala's implicit return type syntax for functions.

Other ways to write this function

This function body is a short way of saying that it returns `true` if the `Int` it is given is an even number, otherwise it returns `false`. If you don't like the way that code reads, it may help to put curly braces around the function body:

```
val isEven = (i: Int) => { i % 2 == 0 }
```

Or you can make the if/else condition more obvious:

```
val isEven = (i: Int) => if (i % 2 == 0) true else false
```

You can also put curly braces around that function body:

```
val isEven = (i: Int) => { if (i % 2 == 0) true else false }
```

Finally, if you prefer a *really* long form, you can write `isEven` like this:

```
val isEven = (i: Int) => {
    if (i % 2 == 0) {
        true
    } else {
        false
    }
}
```

Note: I only show this last version to show an example of a multi-line function body. I don't recommend writing short functions like this.

If you were going to explain any of these functions to another person, a good explanation would be like this:

> "The function `isEven` transforms the input `Int` into a `Boolean` value based on its algorithm, which in this case is `i % 2 == 0`."

> When you read that sentence, it becomes clear that the `Boolean` return value is implied (implicit). I know that when I look at the code I have to pause for a moment before thinking, "Ah, it has a `Boolean` return type," because it takes a moment for my brain to evaluate the function body to determine its return type. Therefore, even though it's more verbose, I generally prefer to write functions that explicitly specify their return type, because then I don't have to read the function body to determine the return type.
>
> IMHO, if (a) you have to read a function's body to determine its return type while (b) what you're really trying to do is understand some other block of code — such as when you're debugging a problem — then (c) this forces you to think about low-level details that aren't important to the problem at hand. That's just my opinion, but it's what I have come to believe; I'd rather just glance at the function's type signature.
>
> Put another way, it's often easier to *write* functions that don't declare their return types, but it's harder to *maintain* them.

The general implicit `val` function syntax

You can come to understand the implicit `val` function syntax by pasting a few functions into the Scala REPL. For instance, when you paste this function into the REPL:

```
val isEven = (i: Int) => i % 2 == 0
```

you'll see that the REPL responds by showing that `isEven` is an instance of something called `<function1>`:

```
scala> val isEven = (i: Int) => i % 2 == 0
isEven: Int => Boolean = <function1>
```

And when you paste a function that takes two input parameters into the REPL:

```
val sum = (a: Int, b: Int) => a + b
```

you'll see that it's an instance of `<function2>`:

```
scala> val sum = (a: Int, b: Int) => a + b
sum: (Int, Int) => Int = <function2>
```

When I line up the REPL output for those two examples, like this:

```
isEven:  Int        => Boolean  =  <function1>
sum:     (Int, Int) => Int      =  <function2>
```

you can begin to see that the general form for the way the REPL displays function variables is this:

```
variableName: type = value
```

You can see this more clearly when I highlight the function types and values. This is the REPL output for `isEven`:

```
isEven: Int => Boolean = <function1>
------  --------------   -----------
name         type            value
```

and this is the output for the `sum` function:

```
sum:   (Int, Int) => Int  =  <function2>
----   -----------------     -----------
name         type                value
```

The type of the `isEven` function can be read as, "Transforms an `Int` value into a `Boolean` value," and the `sum` function can be read as, "Takes two `Int` input parameters and transforms them into an `Int`."

> Cool FP developers generally don't say, "a function returns a result." They say things like, "a function transforms its inputs into an output value." Or, as it's stated in the Land of Lisp book, Lisp purists prefer to say that "a function *evaluates* to a result." This may seem like a minor point, but I find that using phrases like this helps my brain to think of my code as being a combination of algebraic functions (or equations) — and that's a good way to think.

What `<function1>` and `<function2>` mean

In the "Explaining the `val` Function Syntax" appendix I write more about this topic, but in short, the output `<function1>` indicates that `isEven` is an instance of the `Function1` trait (meaning that it has one input parameter), and `<function2>` means that `sum` is an instance of the `Function2` trait (meaning that it has two input parameters). The actual "value" of a function is the full body of the function, but rather than show all of that, the REPL uses `<function1>` and `<function2>` to show that `isEven` and `sum` are instances of these types.

As I discuss in that appendix, behind the scenes the Scala compiler converts this function:

```
val sum = (a: Int, b: Int) => a + b
```

into code that looks a lot like this:

```
val sum = new Function2[Int, Int, Int] {
    def apply(a: Int, b: Int): Int = a + b
}
```

I don't want to get too hung up on these details right now, but this is where the `Function2` reference comes from. For more information on this topic, see the "Explaining the `val` Function Syntax" appendix.

Passing functions into other functions

A great thing about functional programming is that you can pass functions around just like other variables, and the most obvious thing this means is that you can pass one function into another. A good way to demonstrate this is with the methods in the Scala collections classes.

For example, given this list of integers (`List[Int]`):

```
val ints = List(1,2,3,4)
```

and these two functions that take `Int` parameters:

```
val isEven = (i: Int) => i % 2 == 0
val double = (i: Int) => i * 2
```

you can see that `isEven` works great with the `List` class `filter` method:

```
scala> ints.filter(isEven)
res0: List[Int] = List(2, 4)
```

and the `double` function works great with the `map` method:

```
scala> ints.map(double)
res1: List[Int] = List(2, 4, 6, 8)
```

Passing functions into other functions like this is what functional programming is all about.

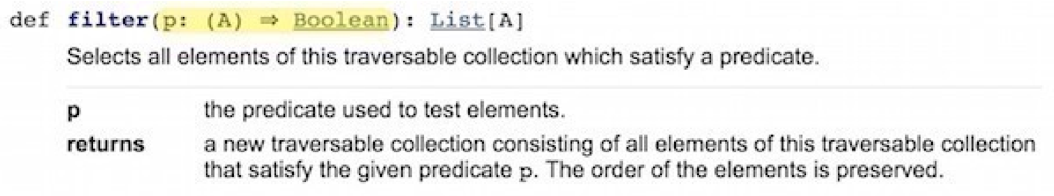
How this works (the short answer)

In the upcoming lessons on Higher-Order Functions I show how to *write* methods like `map` and `filter`, but here's a short discussion of how the process of passing one function into another function (or method) works.

Technically `filter` is written as a method that takes a function as an input parameter. Any function it accepts must (a) take an element of the *type* contained in the collection, and (b) return a `Boolean` value. Because in this example `filter` is invoked on `ints` — which is a `List[Int]` — it expects a function that takes an `Int` and returns a `Boolean`. Because `isEven` transforms an `Int` to a `Boolean`, it works great with `filter` for this collection.

A look at the Scaladoc

The `filter` method Scaladoc is shown in Figure 22.2. Notice how it takes a predicate which has the generic type `A` as its input parameter, and it returns a `List` of the same generic type `A`. It's defined this way because `filter` doesn't *transform* the list elements, it just filters out the ones you don't want.



```
def filter(p: (A) ⇒ Boolean): List[A]
```

Selects all elements of this traversable collection which satisfy a predicate.

p	the predicate used to test elements.
returns	a new traversable collection consisting of all elements of this traversable collection that satisfy the given predicate p. The order of the elements is preserved.

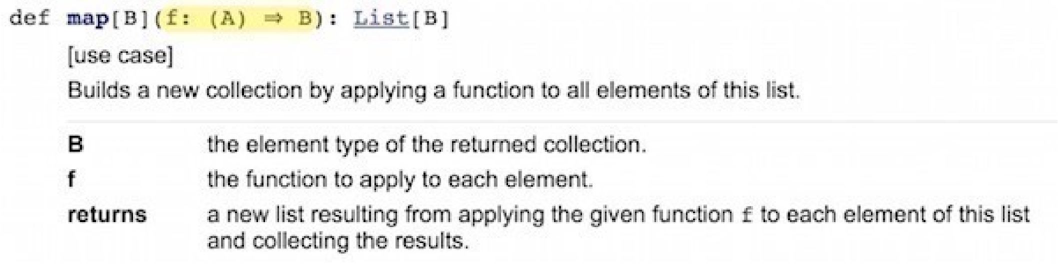
Figure 22.2: The `filter` *method of Scala's* `List` *class.*

As shown in Figure 22.3, `map` also takes a function that works with generic types. In my example, because `ints` is a `List[Int]`, you can think of the generic type `A` in the image as an `Int`. Because `map` is intended to let you *transform* data, the generic type `B` can be any type. In my example, `double` is a function that takes an `Int` and returns an `Int`, so it works great with `map`.

I explain this in more detail in upcoming lessons, but the important point for this lesson is that you *can* pass a function variable into another function.

Because functions are variables …

Because functions are variables, you can do all sorts of things with them. For instance, if you define two functions like this:



```
def map[B](f: (A) ⇒ B): List[B]
```

[use case]

Builds a new collection by applying a function to all elements of this list.

B	the element type of the returned collection.
f	the function to apply to each element.
returns	a new list resulting from applying the given function `f` to each element of this list and collecting the results.

Figure 22.3: The `map` *method of Scala's* `List` *class.*

```
val double = (i: Int) => i * 2
val triple = (i: Int) => i * 3
```

you can have fun and store them in a `Map`:

```
val functions = Map(
    "2x" -> double,
    "3x" -> triple
)
```

If you put that code into the REPL, you'll have two functions stored as values inside a `Map`.

Now that they're in there, you can pass the `Map` around as desired, and then later on get references to the functions using the usual `Map` approach, i.e., by supplying their key values. For example, this is how you get a reference to the `double` function that's stored in the `Map`:

```
scala> val dub = functions("2x")
d: Int => Int = <function1>
```

This is just like getting a `String` or an `Int` or any other reference out of a `Map` — you specify the key that corresponds to the value.

Now that you have a reference to the original `double` function, you can invoke it:

```
scala> dub(2)
res0: Int = 4
```

You can do the same things with the other function I put in the `Map`:

```
scala> val trip = functions("3x")
t: Int => Int = <function1>

scala> trip(2)
res1: Int = 6
```

These examples show how to create functions as variables, store them in a `Map`, get them out of the `Map`, and then use them.

The point of this example

Besides showing how to put function variables into `Maps`, a key point of this example is: in Scala you can use a function variable just like a `String` variable or an `Int` variable. The sooner you begin treating functions as variables in your own code, the further you'll be down the path of becoming a great functional programmer.

Exercise

Given what I've shown so far, this request may be a bit of an advanced exercise, but … here's that `Map` example again:

```
val functions = Map(
    "2x" -> double,
    "3x" -> triple
)
```

Given that `Map`, sketch its data type here:

As an example of what I'm looking for, this `Map`:

```
val m = Map("age" -> 42)
```

has a data type of:

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

That's what I'm looking for in this exercise: the *type* of the `Map` named `functions`.

Solution to the exercise

If you pasted the `Map` code into the REPL, you saw its output:

```
Map[String, Int => Int] = Map(2x -> <function1>, 3x -> <function1>)
```

The first part of that output shows the `Map`'s data type:

```
Map[String, Int => Int]
```

The data type for the `Map`'s *key* is `String`, and the type for its *value* is shown as `Int => Int`. That's how you write the *type* for a function that transforms a single `Int` input parameter to a resulting `Int` value. As you know from the previous discussion, this means that it's an instance of the `Function1` trait.

As a second example, if the `Map` was holding a function that took two `Int`'s as input parameters and returns an `Int` — such as the earlier `sum` function — its type would be shown like this:

```
Map[(Int, Int) => Int]
```

That would be a `Function2` instance, because it takes two input parameters.

Examples of `val` functions

To help you get comfortable with the "implicit return type" version of the `val` function syntax, here are the functions I showed in this lesson:

```
val isEven = (i: Int) => i % 2 == 0
val sum = (a: Int, b: Int) => a + b
val double = (i: Int) => i * 2
val triple = (i: Int) => i * 3
```

And here are a few more functions that show different input parameter types:

```
val strlen = (s: String) => s.length
val concat = (a: String, b: String) => a + b

case class Person(firstName: String, lastName: String)
val fullName = (p: Person) => s"${p.firstName} ${p.lastName}"
```

Summary

Here's a summary of what I showed in this lesson:

- Function literals can be assigned to `val` fields to create function variables
- To be consistent with algebra and other FP resources, I refer to these fields are *variables* rather than *values*
- Examples of the `val` function syntax
- A function is an instance of a `FunctionN` trait, such as `Function1` or `Function2`
- What various function type signatures look like in the REPL
- How to pass a function into another function
- How to treat a function as a variable by putting it in a `Map`
- That, in general, you can use a function variable just like any other variable

In regards to `val` function signatures, understanding them is important because you'll see them in many places, including function literals, the Scaladoc, REPL output, and other developer's code. You'll also need to know this syntax so you can write your own functions that take other functions as input parameters.

What's next

The next lesson shows that you can use `def` methods just like `val` functions. That's important because most developers prefer to use the `def` method syntax to define their algorithms.

See also

- Scala's Function1 trait

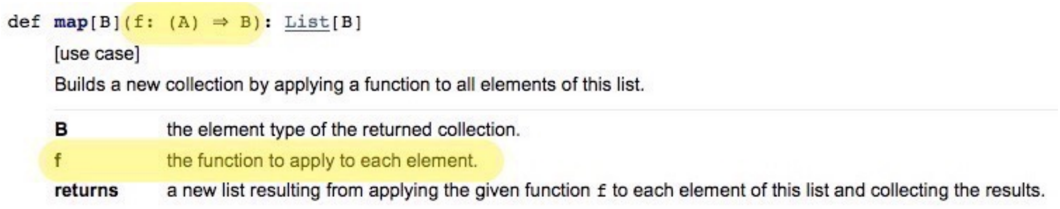
23

Using Methods As If They Were Functions

> *“The owls are not what they seem.”*
>
> From the television series, *Twin Peaks*

Goals

As shown in Figure 23.1, have you noticed that the Scaladoc for the `List` class `map` method clearly shows that it takes a *function*?



```
def map[B](f: (A) ⇒ B): List[B]
```

[use case]

Builds a new collection by applying a function to all elements of this list.

B	the element type of the returned collection.
f	the function to apply to each element.
returns	a new list resulting from applying the given function `f` to each element of this list and collecting the results.

Figure 23.1: The `map` *method of Scala’s* `List` *class.*

But despite that, you can somehow pass it a *method*, and it still works, as shown in this code:

```
// [1] create a method
scala> def doubleMethod(i: Int) = i * 2
doubleMethod: (i: Int)Int

// [2] supply the method where a function is expected
scala> List(1,2,3).map(doubleMethod)
res0: List[Int] = List(2, 4, 6)
```

The intent of this lesson is to provide a brief explanation of how this works, and because it works, how it affects your Scala/FP code.

> I only cover this topic lightly in this lesson. If you want more details after reading this lesson, see the appendix, "The Differences Between `val` and `def` When Creating Functions."

Motivation

I think it's safe to say that most Scala/FP developers prefer to define their "functions" using the `def` keyword. Although the result isn't 100% exactly the same as writing a `val` function, Scala lets you treat both approaches the same, such as when you pass a `def` method into another function. Therefore, because the syntax of `def` methods seems to be more comfortable for developers to read and write, most developers use the `def` approach.

A `def` method is not a `val` (Part 1)

From the previous lessons, you know that this `val isEven` example is an instance of the `Function1` trait:

```
scala> val isEven = (i: Int) => i % 2 == 0
isEven: Int => Boolean = <function1>
```

However, when you write the same algorithm using `def`, the REPL output shows that you have created something else:

```
scala> def isEven(i: Int) = i % 2 == 0
isEven: (i: Int)Boolean
```

The REPL output for the two examples is clearly different. This is because a `val` function is an instance of a `Function0` to `Function22` trait, but a `def` method is … well … when you're not working in the REPL — when you're writing a real application — it's a method that needs to be defined inside of a `class`, `object`, or `trait`.

A deeper look

While this reality is "fudged" a little bit inside the REPL, when you are writing Scala code in a real application, that statement is correct: the only way you can define `def` methods is within a `class`, `object`, or `trait`.

You can easily demonstrate the differences. First, create a file named *Methods.scala* and put this code in it:

```
class Methods {
    def sum(a: Int, b: Int) = a + b
}
```

If you compile that code with `scalac`:

```
$ scalac Methods.scala
```

and then run `javap` on the resulting *Methods.class* file you'll see this output:

```
$ javap Methods
Compiled from "Methods.scala"
public class Methods {
    public int sum(int, int);
    public Methods();
}
```

`sum` is clearly a method in the class named `Methods`. Conversely, if you create a `sum2` *function* in that same class, like this:

```
class Methods {
    def sum(a: Int, b: Int) = a + b
    val sum2 = (a: Int, b: Int) => a + b
}
```

and then compile it with `scalac` and examine the bytecode again with `javap`, you'll see that a `val` function creates something completely different:

```
public scala.Function2<java.lang.Object, java.lang.Object, java.lang.Object> sum2();
```

This lesson explores these differences, particularly from the point of view of using `def` methods just as though they are functions.

A `def` method is not a `val` (Part 2)

In addition to showing that `def` methods are different than `val` functions, the REPL also shows that a method is not a variable that you can pass around. That is, you know that you can assign an `Int` to a variable name:

```
scala> val x = 1
x: Int = 1
```

and then show information about that variable:

```
scala> x
res0: Int = 1
```

You can also define a *function* and assign it to a variable:

```
scala> val double = (i: Int) => i * 2
double: Int => Int = <function1>
```

and then show information about it:

```
scala> double
res1: Int => Int = <function1>
```

But if you define a method using `def`:

```
scala> def triple(i: Int) = i * 3
triple: (i: Int)Int
```

and then try to show that method's "variable," what you'll actually get is an error:

```
scala> triple
<console>:12: error: missing arguments for method triple;
follow this method with `_' if you want to treat it as a partially applied function
       triple
       ^
```

The REPL shows this error because the `triple` method is not a variable (field name) in the same way that an `Int` or a function is a variable.

> Not yet, anyway. Very shortly I'll demonstrate how you can *manually* create a variable from a method.

Recap

The reason I show these examples is to demonstrate that until you do something like passing a method into a function, a `def` method is not the same as a `val` function. Despite that, we know that somehow you *can* later treat a method as a function.

Which leads to the next question …

How is it that I can use a method like a function?

In the appendix, "The Differences Between `val` and `def` When Creating Functions," I show in detail how the Scala compiler lets you use `def` methods just like `val` functions. Without repeating too much of that information here, you'll find that the solution is hinted at in Version 2.9 of *The Scala Language Specification*:

> "*Eta-expansion* converts an expression of *method* type to an equivalent expression of *function* type."

What that means is that when the Scala compiler is given these two lines of code:

```
def isEven(i: Int) = i % 2 == 0   // define a method
val evens = nums.filter(isEven)   // pass the method into a function
```

it uses this "Eta Expansion" capability to automatically convert the *method* `isEven` into a *function* — a true `Function1` instance — so it can be passed into `filter`.

This happens automatically during the compilation process, so you generally don't even have to think about. In fact, I used Scala for almost a year before I thought, "Hey, how is this even working?"

How to manually convert a method to a function

To give you an idea of how Eta Expansion works, let's use the earlier `triple` example. I first defined this method:

```
scala> def triple(i: Int) = i * 3
triple: (i: Int)Int
```

and then when I tried to show its value in the REPL, I got this error:

```
scala> triple
<console>:12: error: missing arguments for method triple;
follow this method with `_' if you want to treat it as a
partially applied function
       triple
       ^
```

The error message states that you can follow this method with an underscore to treat the method as a *partially applied function*. That is true, and I demonstrate it in the next lesson. But for this lesson, the important thing to know is that *doing this creates a function from your method*.

To demonstrate this, go ahead and do what the error message says. Follow the method name with an underscore, and also assign that result to a variable name:

```
scala> val tripleFn = triple _
tripleFn: Int => Int = <function1>
```

Notice that the signature of this result is `Int => Int`. This means that `tripleFn` is a function that takes one `Int` as an input parameter, and returns an `Int` result. The REPL output also shows that `tripleFn` has a value `<function1>`, which means that it's an instance of the `Function1` trait. Because it's now a real function, you can display its value in the REPL:

```
scala> tripleFn
res0: Int => Int = <function1>
```

This new function works just like the method works, taking an `Int` input parameter and returning an `Int` result:

```
scala> tripleFn(1)
res0: Int = 3
```

To confirm that this manually-created function works as advertised, you can pass it into the `map` method of a `List[Int]`, which really does expect a function, not a method:

```
// create a List[Int]
scala> val x = List(1,2,3)
x: List[Int] = List(1, 2, 3)

// pass in the `tripleFn` function
scala> x.map(tripleFn)
res1: List[Int] = List(3, 6, 9)
```

This is a short example of what Eta Expansion does for you behind the scenes, during the compilation process.

To sum up this point, this process happens automatically when you pass a `def` method into a function that expects a *function*. It also lets you use `def` methods just like they are functions in many other situations.

> For much more information on this process, see the appendix, "The Differences Between `val` and `def` When Creating Functions."

> It’s hard to really “prove” in the REPL that this is what happens because I don’t know of any way to disable Eta Expansion. But if you could disable it, you would find that the *method* would not work with `map`, and the *function* would work with it.
>
> While you can’t prove it in the REPL, you can show what happens behind the scenes with the Scala compiler. If you start with this class:
>
> ```
> class EtaExpansionTest {
>
>     def double(i: Int) = i * 2
>
>     def foo = {
>         val xs = List(1,2,3)
>         xs.map(double)     // pass the `double` method into `map`
>     }
> }
> ```
>
> and then compile it with this command:
>
> ```
> $ scalac -Xprint:all Methods.scala
> ```
>
> you’ll see a *lot* of output, and if you take the time to dig through that output, you’ll be amazed at what the compiler does to the `xs.map(double)` code by the time it’s done with it. I won’t go into all of that here, but if you’re interested in how this process works, I encourage you to dig into that output.

In some places it doesn’t happen automatically

In the previous lesson I showed that you can define functions and then store them in a `Map`. Can you do the same thing with methods?

Well, if you define two methods like this:

```
def double(i: Int) = i * 2
def triple(i: Int) = i * 3
```

and then try to store them in a `Map`, like this:

```
val functions = Map(
    "2x" -> double,
    "3x" -> triple
)
```

you'll get the following error messages:

```
<console>:13: error: missing arguments for method double;
follow this method with `_' if you want to treat it as a
partially applied function
           "2x" -> double,
                       ^
<console>:14: error: missing arguments for method triple;
follow this method with `_' if you want to treat it as a
partially applied function
           "3x" -> triple
                       ^
```

Before this lesson those errors might have been a head-scratcher, but now you know how to solve this problem — how to manually convert the methods into functions by following the *method* invocations with an underscore:

```
val functions = Map(
    "2x" -> double _,
    "3x" -> triple _
)
```

That syntax converts the `double` and `triple` *methods* into *functions*, and then everything works as shown in the previous lesson, which in this case means that you can get a function back out of the `Map` and use it:

```
scala> val dub = functions("2x")
dub: Int => Int = <function1>
```

```
scala> dub(3)
res0: Int = 6
```

Why this lesson is important

The reason I showed everything in this lesson is because most developers prefer the `def` method syntax over the `val` function syntax. That is, given the choice to write an algorithm using either approach, developers seem to prefer the `def` approach, and I believe that's because the `def` syntax is easier to read.

Because of this, in the rest of this book I will often write `def` methods and refer to them as functions. Technically this isn't accurate, but because (a) methods can be used just like functions, and (b) I don't want to have to keep writing, "*A method that acts like a function*," I will now start using this terminology.

Summary

Here's a summary of what I showed in this lesson:

- The Scaladoc for collections methods like `map` and `filter` show that they take *functions* as input parameters.
- Despite that, somehow you can pass *methods* into them.
- The reason that works is called "Eta Expansion."
- I showed how to manually convert a method to a function (using the partially-applied function approach).
- As a result of Eta Expansion, you can use `def` to define methods, and then generally treat them in the same way that you use `val` functions.

In this lesson I only covered the basics of how a "`def` method" is like a "`val` function." For more details on the differences, see the appendix, "The Differences Between `val` and `def` When Creating Functions."

What’s next

In this lesson I showed that you can generally treat a `def` method just like a `val` function, and not have to worry about the differences between the two. I also showed that if the compiler doesn’t take care of that process for you automatically, you can handle it manually.

In the next lesson you’ll see how to write functions that take other functions as input parameters. With this background, you know that this also means that those functions will be able to take methods as input parameters as well.

24

How to Write Functions That Take Functions as Input Parameters

> *"Haskell functions can take functions as parameters and return functions as return values. A function that does either of those is called a higher order function. Higher order functions aren't just a part of the Haskell experience, they pretty much are the Haskell experience."*
>
> From Learn You a Haskell for Great Good!

Motivation and Goals

The topic I'm about to cover is a big part of functional programming: *power programming* that's made possible by passing functions to other functions to get work done.

So far I've shown I've shown how to be the *consumer* of functions that take other functions as input parameters, that is, the consumer of *Higher Order Functions* (HOFs) like `map` and `filter`. In this lesson I'm going to show everything you need to know to be the *producer* of HOFs, i.e., the writer of HOF APIs.

Therefore, the primary goal of this lesson is to show how to write functions that take other functions as input parameters. I'll show:

- The syntax you use to define function input parameters
- Many examples of that syntax
- How to execute a function once you have a reference to it

As a beneficial side effect of this lesson, you'll be able to read the source code and Scaladoc for other HOFs, and you'll be able to understand the function signatures they're looking for.

Terminology

Before we start, here are a few notes about the terminology I'll use in this lesson.

1) I use the acronym "FIP" to stand for "function input parameter." This isn't an industry standard, but because I use the term so often, I think the acronym makes the text easier to read.
2) As shown already, I'll use "HOF" to refer to "Higher Order Function."
3) As shown in the previous lessons you can create functions as variables, and because of Eta Expansion you can do that by writing them as either (a) `val` functions or (b) `def` methods. Because of this, and because I think `def` methods are easier to read, from now on I'll write `def` methods and refer to them as "functions," even though that terminology isn't 100% accurate.

Introduction

I finished the previous lesson by showing a few function definitions like this:

```
def isEven(i: Int) = i % 2 == 0
def sum(a: Int, b: Int) = a + b
```

I also showed that `isEven` works great when you pass it into the `List` class `filter` method:

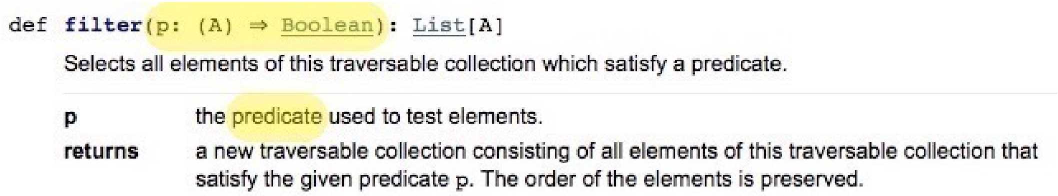
```
scala> val list = List.range(0, 10)
list: List[Int] = List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> val evens = list.filter(isEven)
evens: List[Int] = List(0, 2, 4, 6, 8)
```

The key points of this are:

- The `filter` method accepts a function as an input parameter.
- The functions you pass into `filter` must match the type signature that `filter` expects — in this case creating a function like `isEven` that takes an `Int` as an input parameter and returns a `Boolean`.

Understanding `filter`*'s Scaladoc*

The Scaladoc shows the type of functions `filter` accepts, which you can see in Figure 24.1.



```
def filter(p: (A) ⇒ Boolean): List[A]
```

Selects all elements of this traversable collection which satisfy a predicate.

p	the predicate used to test elements.
returns	a new traversable collection consisting of all elements of this traversable collection that satisfy the given predicate `p`. The order of the elements is preserved.

Figure 24.1: The Scaladoc shows the type of functions `filter` *accepts.*

The Scaladoc text shows that `filter` takes a *predicate*, which is just a function that returns a `Boolean` value.

This part of the Scaladoc:

```
p: (A) => Boolean
```

means that `filter` takes a function input parameter which it names `p`, and `p` must transform a generic input `A` to a resulting `Boolean` value. In my example, where `list` has the type `List[Int]`, you can replace the generic type `A` with `Int`, and read that signature like this:

```
p: (Int) => Boolean
```

Because `isEven` has this type — it transforms an input `Int` into a resulting `Boolean` — it can be used with `filter`.

A lot of functionality with a little code

The `filter` example shows that with HOFs you can accomplish a lot of work with a little bit of code. If `List` didn't have the `filter` method, you'd have to write a custom method like this to do the same work:

```
// what you'd have to do if `filter` didn't exist
def getEvens(list: List[Int]): List[Int] = {
    val tmpArray = ArrayBuffer[Int]()
    for (elem <- list) {
        if (elem % 2 == 0) tmpArray += elem
    }
    tmpArray.toList
}

val result = getEvens(list)
```

Compare all of that imperative code to this equivalent functional code:

```
val result = list.filter(_ % 2 == 0)
```

As you can see, this is a great advantage of functional programming. The code is much more concise, and it's also easier to comprehend.

> As FP developers like to say, you don't tell the computer specifically "how" to do something — you don't specify the nitty-gritty details. Instead, in your FP code you express a thought like, "I want to create a filtered version of this list with this little algorithm." When you do that, and you have good FP language to work with, you write your code at a much higher programming level.

"Common control patterns"

In many situations Scala/FP code can be easier to understand than imperative code. That's because a great benefit of Scala/FP is that methods like `filter`, `map`, `head`, `tail`, etc., are all standard, built-in functions, so once you learn them you don't have to write custom `for` loops any more. As an added benefit, you also don't have to read other developers' custom `for` loops.

> I feel like I say this a lot, but we humans can only keep so much in our brains at one time. Concise, readable code is simpler for your brain and better for your productivity.

I know, I know, when you first come to Scala, all of these methods on the collections classes don't feel like a benefit, they feel overwhelming. But once you realize that almost *every* `for` loop you've ever written falls into neat categories like `map`, `filter`, `reduce`, etc., you also realize what a great benefit these methods are. (And you'll reduce the amount of custom `for` loops you write by at least 90%.)

Here's what Martin Odersky wrote about this in his book, Programming in Scala:

> "You can use functions within your code to factor out common control patterns, and you can take advantage of higher-order functions in the Scala library to reuse control patterns that are common across all programmers' code."

Given this background and these advantages, let's see how to write functions that take other functions as input parameters.

Defining functions that take functions as parameters

To define a function that takes another function as an input parameter, all you have to do is define the signature of the function you want to accept.

To demonstrate this, I'll define a function named `sayHello` that takes a function as an input parameter. I'll name the input parameter `callback`, and also say that `callback` must have no input parameters and must return nothing. This is the Scala syntax to make this happen:

```
def sayHello(callback: () => Unit) {
    callback()
}
```

In this code, `callback` is an input parameter, and more specifically it is a *function input parameter* (or FIP). Notice how it's defined with this syntax:

```
callback: () => Unit
```

Here's how this works:

- `callback` is the name I give to the input parameter. In this case `callback` is a function I want to accept.
- The `callback` signature specifies the *type* of function I want to accept.
- The `()` portion of `callback`'s signature (on the left side of the => symbol) states that it takes no input parameters.
- The `Unit` portion of the signature (on the right side of the => symbol) indicates that the `callback` function should return nothing.
- When `sayHello` is called, its function body is executed, and the `callback()` line inside the body invokes the function that is passed in.

Figure 24.2 reiterates those points.

Now that I've defined `sayHello`, I'll create a function to match `callback`'s signature so I can test it. The following function takes no input parameters and returns nothing, so it matches `callback`'s type signature:

```
def helloAl(): Unit = { println("Hello, Al") }
```

Because the signatures match, I can pass `helloAl` into `sayHello`, like this:

```
sayHello(helloAl)
```

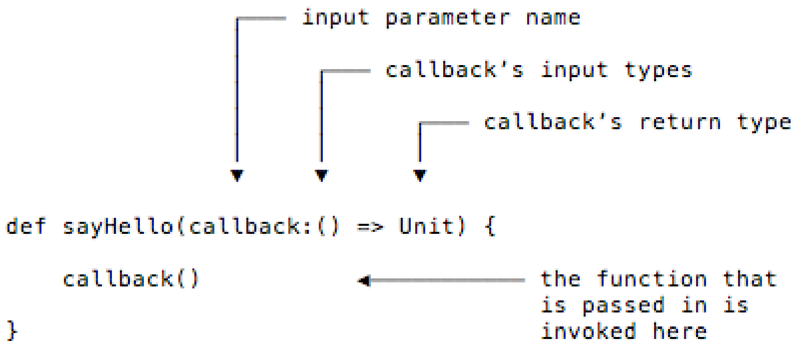


Figure 24.2: How `sayHello` *and* `callback` *work.*

The REPL demonstrates how all of this works:

```
scala> def sayHello(callback:() => Unit) {
     |     callback()
     | }
sayHello: (callback: () => Unit)Unit

scala> def helloAl(): Unit = { println("Hello, Al") }
helloAl: ()Unit

scala> sayHello(helloAl)
Hello, Al
```

If you've never done this before, congratulations. You just defined a function named `sayHello` that takes another function as an input parameter, and then invokes that function when it's called.

It's important to know that the beauty of this approach is not that `sayHello` can take *one* function as an input parameter; the beauty is that it can take *any* function that matches `callback`'s signature. For instance, because this next function takes no input parameters and returns nothing, it also works with `sayHello`:

```
def holaLorenzo(): Unit = { println("Hola, Lorenzo") }
```

Here it is in the REPL:

```
scala> sayHello(holaLorenzo)
Hola, Lorenzo
```

This is a good start. Let's build on it by defining functions that can take more complicated functions as input parameters.

The general syntax for defining function input parameters

I defined `sayHello` like this:

```
def sayHello(callback: () => Unit)
```

Inside of that, the `callback` function signature looks like this:

```
callback: () => Unit
```

I can explain this syntax by showing a couple of examples. Imagine that we're defining a new version of `callback`, and this new version takes a `String` and returns an `Int`. That signature would look like this:

```
callback: (String) => Int
```

Next, imagine that you want to create a different version of `callback`, and this one should take two `Int` parameters and return an `Int`. Its signature would look like this:

```
callback: (Int, Int) => Int
```

As you can infer from these examples, the general syntax for defining function input parameter type signatures is:

```
variableName: (parameterTypes ...) => returnType
```

With `sayHello`, this is how the values line up:

General	`sayHello`	Notes
variableName	`callback`	The name you give the FIP
parameterTypes	()	The FIP takes no input parameters
returnType	`Unit`	The FIP returns nothing

Naming your function input parameters

I find that the parameter name `callback` is good when you first start writing HOFs. Of course you can name it anything you want, and other interesting names at first are `aFunction`, `theFunction`, `theExpectedFunction`, or maybe even `fip`. But, from now on, I'll make this name shorter and generally refer to the FIPs in my examples as just `f`, like this:

```
sayHello(f: () => Unit)
foo(f:(String) => Int)
bar(f:(Int, Int) => Int)
```

Looking at some function signatures

Using this as a starting point, let's look at signatures for some more FIPs so you can see the differences. To get started, here are two signatures that define a FIP that takes a `String` and returns an `Int`:

```
sampleFunction(f: (String) => Int)
sampleFunction(f: String => Int)
```

The second line shows that when you define a function that takes only one input parameter, you can leave off the parentheses.

Next, here's the signature for a function that takes two `Int` parameters and returns an `Int`:

```
sampleFunction(f: (Int, Int) => Int)
```

Can you imagine what sort of function matches that signature?

(A brief pause here so you can think about that.)

Any function that takes two `Int` input parameters and returns an `Int` matches that signature, so functions like these all fit:

```
def sum(a: Int, b: Int): Int = a + b
def product(a: Int, b: Int): Int = a * b
def subtract(a: Int, b: Int): Int = a - b
```

You can see how `sum` matches up with the FIP signature in Figure 24.3.

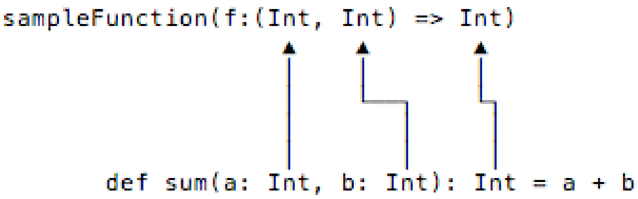


Figure 24.3: How `sum` *matches up with the parameters in the FIP signature.*

For me, an important part of this is that no matter how complicated the type signatures get, they always follow the same general syntax I showed earlier:

```
variableName: (parameterTypes ...) => returnType
```

For example, all of these FIP signatures follow the same pattern:

```
f: () => Unit
f: String => Int
f: (String) => Int
f: (Int, Int) => Int
f: (Person) => String
f: (Person) => (String, String)
f: (String, Int, Double) => Seq[String]
```

```
f: List[Person] => Person
```

A note about "type signatures"

I'm being a little loose with my verbiage here, so let me tighten it up for a moment. When I say that this is a "type signature":

```
f: String => Int
```

that isn't 100% accurate. The *type signature* is really just this part:

```
String => Int
```

Therefore, being 100% accurate, these are the type signatures I just showed:

```
() => Unit
String => Int
(String) => Int
(Int, Int) => Int
(Person) => String
(Person) => (String, String)
(String, Int, Double) => Seq[String]
List[Person] => Person
```

This may seem like a picky point, but because FP developers talk about type signatures all the time, I want to take that moment to be more precise.

> It's common in FP to think about types a *lot* in your code. You might say that you "think in types."

A function that takes an `Int` parameter

Recapping for a moment, I showed the `sayHello` function, whose `callback` parameter states that it takes no input parameters and returns nothing:

```
sayHello(callback: () => Unit)
```

I refer to `callback` as a FIP, which stands for “function input parameter.”

Now let’s look at a few more FIPs, with each example building on the one before it.

First, here’s a function named `runAFunction` that defines a FIP whose signature states that it takes an `Int` and returns nothing:

```
def runAFunction(f: Int => Unit): Unit = {
    f(42)
}
```

The body says, “Whatever function you give to me, I’m going to pass the `Int` value `42` into it.” That’s not terribly useful or functional, but it’s a start.

Next, let’s define a function that matches `f`’s type signature. The following `printAnInt` function takes an `Int` parameter and returns nothing, so it matches:

```
def printAnInt (i: Int): Unit = { println(i+1) }
```

Now you can pass `printAnInt` into `runAFunction`:

```
runAFunction(printAnInt)
```

Because `printAnInt` is invoked inside `runAFunction` with the value `42`, this prints `43`. Here’s what it all looks like in the REPL:

```
scala> def runAFunction(f: Int => Unit): Unit = {
     |     f(42)
     | }
runAFunction: (f: Int => Unit)Unit

scala> def printAnInt (i: Int): Unit = { println(i+1) }
printAnInt: (i: Int)Unit

scala> runAFunction(printAnInt)
43
```

Here's a second function that takes an `Int` and returns nothing:

```
def plusTen(i: Int) { println(i+10) }
```

When you pass `plusTen` into `runAFunction`, you'll see that it also works, printing `52`:

```
runAFunction(plusTen)   // prints 52
```

The power of the technique

Although these examples don't do too much yet, you can see the power of HOFs:

> You can easily swap in interchangeable algorithms.

As long as the signature of the function you pass in matches the signature that's expected, your algorithms can do anything you want. This is comparable to swapping out algorithms in the OOP Strategy design pattern.

Let's keep building on this…

Taking a function parameter along with other parameters

Here's a function named `executeNTimes` that has two input parameters: a function, and an `Int`:

```
def executeNTimes(f: () => Unit, n: Int) {
    for (i <- 1 to n) f()
}
```

As the code shows, `executeNTimes` executes the `f` function `n` times. To test this, define a function that matches `f`'s signature:

```
def helloWorld(): Unit = { println("Hello, world") }
```

and then pass this function into `executeNTimes` along with an `Int`:

```
scala> executeNTimes(helloWorld, 3)
Hello, world
Hello, world
Hello, world
```

As expected, `executeNTimes` executes the `helloWorld` function three times. Cool.

More parameters, everywhere

Next, here's a function named `executeAndPrint` that takes a function and two `Int` parameters, and returns nothing. It defines the FIP `f` as a function that takes two `Int` values and returns an `Int`:

```
def executeAndPrint(f: (Int, Int) => Int, x: Int, y: Int): Unit = {
    val result = f(x, y)
    println(result)
}
```

`executeAndPrint` passes the two `Int` parameters it's given into the FIP it's given in this line of code:

```
val result = f(x, y)
```

Except for the fact that this function doesn't have a return value, this example shows a common FP technique:

- Your function takes a FIP.
- It takes other parameters that work with that FIP.
- You apply the FIP (`f`) to the parameters as needed, and return a value. (Or, in this example of a function with a side effect, you print something.)

To demonstrate `executeAndPrint`, let's create some functions that match `f`'s signature. Here are a couple of functions take two `Int` parameters and return an `Int`:

```
def sum(x: Int, y: Int) = x + y
def multiply(x: Int, y: Int) = x * y
```

Now you can call `executeAndPrint` with these functions as the first parameter and whatever `Int` values you want to supply as the second and third parameters:

```
executeAndPrint(sum, 3, 11)       // prints 14
executeAndPrint(multiply, 3, 9)   // prints 27
```

Let's keep building on this…

Taking multiple functions as input parameters

Now let's define a function that takes multiple FIPs, and other parameters to feed those FIPs. Let's define a function like this:

- It takes one function parameter that expects two `Int`s, and returns an `Int`
- It takes a second function parameter with the same signature
- It takes two other `Int` parameters
- The `Int`s will be passed to the two FIPs
- It will return the results from the first two functions as a tuple — a `Tuple2`, to be specific

Since I learned FP, I like to think in terms of "Function signatures first," so here's a function signature that matches those bullet points:

```
def execTwoFunctions(f1:(Int, Int) => Int,
                     f2:(Int, Int) => Int,
                     a: Int,
                     b: Int): Tuple2[Int, Int] = ???
```

Given that signature, can you imagine what the function body looks like?

(I'll pause for a moment to let you think about that.)

Here's what the complete function looks like:

```
def execTwoFunctions(f1: (Int, Int) => Int,
                     f2: (Int, Int) => Int,
                     a: Int,
                     b: Int): Tuple2[Int, Int] = {
    val result1 = f1(a, b)
    val result2 = f2(a, b)
    (result1, result2)
}
```

That's a verbose (clear) solution to the problem. You can shorten that three-line function body to just this, if you prefer:

```
(f1(a,b), f2(a,b))
```

Now you can test this new function with the trusty `sum` and `multiply` functions:

```
def sum(x: Int, y: Int) = x + y
def multiply(x: Int, y: Int) = x * y
```

Using these functions as input parameters, you can test `execTwoFunctions`:

```
val results = execTwoFunctions(sum, multiply, 2, 10)
```

The REPL shows the results:

```
scala> val results = execTwoFunctions(sum, multiply, 2, 10)
results: (Int, Int) = (12,20)
```

I hope this gives you a taste for not only how to write HOFs, but the power of using them in your own code.

Okay, that's enough examples for now. I'll cover two more topics before finishing this lesson, and then in the next lesson you can see how to write a `map` function with everything I've shown so far.

The FIP syntax is just like the `val` function syntax

A nice thing about Scala is that once you know how things work, you can see the consistency of the language. For example, the syntax that you use to define FIPs is the same as the "explicit return type" (ERT) syntax that you use to define functions.

> I show the ERT syntax in detail in the "Explaining Scala's `val` Function Syntax" appendix.

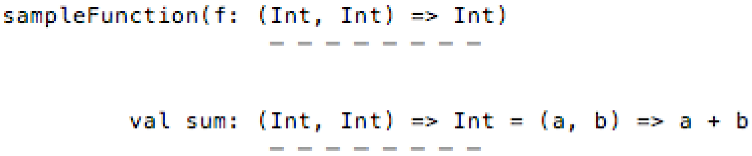
What I mean by this is that earlier I defined this function:

```
sampleFunction(f: (Int, Int) => Int)
```

The part of this code that defines the FIP signature is exactly the same as the ERT signature for the `sum` function that I define in the `val` Function Syntax appendix:

```
val sum: (Int, Int) => Int = (a, b) => a + b
```

You can see what I mean if you line the two functions up, as shown in Figure 24.4.



```
sampleFunction(f: (Int, Int) => Int)

           val sum: (Int, Int) => Int = (a, b) => a + b
```

Figure 24.4: The FIP signature is exactly the same as the ERT signature for the `sum` *function.*

Once you understand the FIP type signature syntax, it becomes easier to read things like the ERT function syntax and the Scaladoc for HOFs.

The general thought process of designing HOFs

Personally, I'm rarely smart enough to see exactly what I want to do with all of my code beforehand. Usually I *think* I know what I want to do, and then as I start coding I realize that I really want something else. As a result of this, my usual thought process when it comes to writing HOFs looks like this:

1. I write some code
2. I write more code
3. I realize that I'm starting to duplicate code
4. Knowing that duplicating code is bad, I start to refactor the code

Actually, I have this same thought process whether I'm writing OOP code or FP code, but the difference is in *what I do next*.

With OOP, what I might do at this point is to start creating class hierarchies. For instance, if I was working on some sort of tax calculator in the United States, I might create a class hierarchy like this:

```
trait StateTaxCalculator
class AlabamaStateTaxCalculator extends StateTaxCalculator ...
class AlaskaStateTaxCalculator extends StateTaxCalculator ...
class ArizonaStateTaxCalculator extends StateTaxCalculator ...
```

Conversely, in FP, my approach is to first define an HOF like this:

```
def calculateStateTax(f: Double => Double, personsIncome: Double): Double = ...
```

Then I define a series of functions I can pass into that HOF, like this:

```
def calculateAlabamaStateTax(income: Double): Double = ...
def calculateAlaskaStateTax(income: Double): Double = ...
def calculateArizonaStateTax(income: Double): Double = ...
```

As you can see, that's a pretty different thought process.

> Note: I have no idea whether I'd approach these problems *exactly* as shown. I just want to demonstrate the difference in the general thought process between the two approaches, and in that regard — creating a

> class hierarchy versus a series of functions with a main HOF — I think this example shows that.

To summarize this, the thought process, "I need to refactor this code to keep it DRY," is the same in both OOP and FP, but the way you refactor the code is very different.

Summary

A function that takes another function as an input parameter is called a "Higher Order Function," or HOF. This lesson showed how to write HOFs in Scala, including showing the syntax for function input parameters (FIPs) and how to execute a function that is received as an input parameter.

As the lesson showed, the general syntax for defining a function as an input parameter is:

```
variableName: (parameterTypes ...) => returnType
```

Here are some examples of the syntax for FIPs that have different types and numbers of arguments:

```
def exec(f:() => Unit) = ???   // note: i don't show the function body
                             // for any of these examples

def exec(f: String => Int)     // parentheses not needed
def exec(f: (String) => Int)
def exec(f: (Int) => Int)
def exec(f: (Double) => Double)
def exec(f: (Person) => String)
def exec(f: (Int) => Int, a: Int, b: Int)
def exec(f: (Pizza, Order) => Double)
def exec(f: (Pizza, Order, Customer, Discounts) => Currency)
def exec(f1: (Int) => Int, f2:(Double) => Unit, s: String)
```

What’s next

In this lesson I showed how to write HOFs. In the next lesson we’ll put this knowledge to work by writing a complete `map` function that uses the techniques shown in this lesson.

25

How to Write a 'map' Function

> *"He lunged for the maps. I grabbed the chair and hit him with it. He went down. I hit him again to make sure he stayed that way, stepped over him, and picked up the maps."*
>
> Ilona Andrews, Magic Burns

In the previous lesson I showed how to write higher-order functions (HOFs). In this lesson you'll use that knowledge to write a `map` function that can work with a `List`.

Writing a `map` function

Imagine a world in which you know of the concept of "mapping," but sadly a `map` method isn't built into Scala's `List` class. Further imagine that you're not worried about *all* lists, you just want a `map` function for a `List[Int]`.

Knowing that life is better with `map`, you sit down to write your own `map` method.

First steps

As I got better at FP, I came to learn that my first actions in writing most functions are:

1. Accurately state the problem as a sentence
2. Sketch the function signature

I'll follow that approach to solve this problem.

Accurately state the problem

For the first step, I'll state the problem like this:

> I want to write a `map` function that can be used to apply other functions to each element in a `List[Int]` that it's given.

Sketch the function signature

My second step is to sketch a function signature that matches that statement. A blank canvas is always hard to look at, so I start with the obvious; I want a `map` function:

```
def map
```

Looking back at the problem statement, what do I know? Well, first, I know that `map` is going to take a function as an input parameter, and it's also going to take a `List[Int]`. Without thinking too much about the input parameters just yet, I can now sketch this:

```
def map(f: (?) => ?, list: List[Int]): ???
```

Knowing how `map` works, I know that it should return a `List` that contains the same number of elements that are in the input `List`. For the moment, the important part about this is that this means that `map` will return a `List` of some sort:

```
def map(f: (?) => ?, list: List[Int]): List...
                                       ----
```

Given how `map` works — it applies a function to every element in the input list — the *type* of the output `List` can be anything: a `List[Double]`, `List[Float]`, `List[Foo]`, etc. This tells me that the `List` that `map` returns needs to be a generic type, so I add that at the end of the function declaration:

```
def map(f: (?) => ?, list: List[Int]): List[A]
                                       -------
```

Because of Scala's syntax, I need to add the generic type before the function signature as well:

```
def map[A](f: (?) => ?, list: List[Int]): List[A]
       ---
```

Going through that thought process tells me everything I need to know about the signature for the function input parameter `f`:

- Because `f`'s input parameter will come from the `List[Int]`, the parameter type must be `Int`
- Because the overall `map` function returns a `List` of the generic type `A`, `f` must also return the generic type `A`

The first statement lets me make this change to the definition of `f`:

```
def map[A](f: (Int) => ?, list: List[Int]): List[A]
               ---
```

and the second statement lets me make this change:

```
def map[A](f: (Int) => A, list: List[Int]): List[A]
                       -
```

When I define a FIP that has only one input parameter I can leave the parentheses off, so if you prefer that syntax, the finished function signature looks like this:

```
def map[A](f: Int => A, list: List[Int]): List[A]
```

Cool. That seems right. Now let's work on the function body.

The `map` function body

A `map` function works on every element in a list, and because I haven't covered recursion yet, this means that we're going to need a `for` loop to loop over every element in the input list.

Because I know that `map` returns a list that has one element for each element in the input list, I further know that this loop is going to be a for/yield loop without any filters:

```
def map[A](f: (Int) => A, list: List[Int]): List[A] = {
    for {
        x <- list
    } yield ???
}
```

The only question now is, what exactly should the loop *yield*?

(I'll pause for a moment here to let you think about that.)

The answer is that the `for` loop should `yield` the result of applying the input function `f` to the current element in the loop. Therefore, I can finish the `yield` expression like this:

```
def map[A](f: (Int) => A, list: List[Int]): List[A] = {
    for {
        x <- list
    } yield f(x)   //<-- apply 'f' to each element 'x'
}
```

And that is the solution for the problem that was stated.

You can use the REPL to confirm that this solution works as desired. First, paste the `map` function into the REPL. Then create a list of integers:

```
scala> val nums = List(1,2,3)
nums: List[Int] = List(1, 2, 3)
```

Then write a function that matches the signature `map` expects:

```
scala> def double(i: Int): Int = i * 2
double: (i: Int)Int
```

Then you can use `map` to apply `double` to each element in `nums`:

```
scala> map(double, nums)
res0: List[Int] = List(2, 4, 6)
```

The `map` function works.

Bonus: Make it generic

I started off by making `map` work only for a `List[Int]`, but at this point it's easy to make it work for any `List`. This is because there's nothing inside the `map` function body that depends on the given `List` being a `List[Int]`:

```
for {
    x <- list
} yield f(x)
```

That's as "generic" as code gets; there are no `Int` references in there. Therefore, you can make `map` work with generic types by replacing each `Int` reference in the function signature with a generic type. Because this type appears before the other generic type in the function signature, I'll first convert the old `A`'s to `B`'s:

```
def map[B](f: (Int) => B, list: List[Int]): List[B] = ...
        -              -                         -
```

Then I replace the `Int` references with `A`, and put an `A` in the opening brackets, resulting in this signature:

```
def map[A,B](f: (A) => B, list: List[A]): List[B] = {
        -        -                   -
```

If you want to take this even further, there's also nothing in this code that depends on the input "list" being a `List`. Because `map` works its way from the first element in the list to the last element, it doesn't matter if the `Seq` is an `IndexedSeq` or a `LinearSeq`, so you can use the parent `Seq` class here instead of `List`:

```
def map[A,B](f: (A) => B, list: Seq[A]): Seq[B] = {
                                   ---      ---
```

With this new signature, the complete, generic `map` function looks like this:

```
def map[A,B](f: (A) => B, list: Seq[A]): Seq[B] = {
    for {
        x <- list
    } yield f(x)
}
```

I hope you enjoyed that process. It's a good example of how I design functions these days, starting with the signature first, and then implementing the function body.

Exercise: Write a `filter` function

Now that you've seen how to write a `map` function, I encourage you to take the time to write a `filter` function. Because `filter` doesn't return a sequence that's the same size as the input sequence, its algorithm will be a little different, but it still needs to return a sequence in the end.

What's next

While this lesson provided a detailed example of how to write a function that takes other functions as an input parameter, the next lesson will show how to write functions that take "blocks of code" as input parameters. That technique and syntax is similar to what I just showed, but the "use case" for this other technique — known as "by-name parameters" — is a little different.

After that lesson, I'll demonstrate how to combine these techniques with a Scala feature that lets a function have multiple input parameter groups.

26

How to Use By-Name Parameters

> *"Call me, call me by my name, or call me by number."*
>
> Chesney Hawkes, "The One and Only"

Introduction

In previous lessons I showed how to pass a function into another function. I showed *how* to do that (the syntax), and I also showed *why* to do that (to easily pass in new algorithms).

While that's a great feature, sometimes you just want to write a function that takes a more general "block of code." I typically do this when I'm writing a custom control structure, and as it turns out it's also common technique in FP.

In Scala we say that a function that defines an input parameter like this is a "by-name" parameter, which is also referred to as a "call by-name" parameter.

Goals

Given that introduction, my goals for this lesson are to show:

- The differences between by-value and by-name parameters
- The by-name syntax
- How to use by-name parameters
- Examples of when they are appropriate

- A comparison of by-name parameters and higher-order functions

Background: By-value parameters

If you define a `Person` class like this:

```
case class Person(var name: String)
```

and then pass it into a Scala function, it's said to be a "call by-value" argument. You can read much more about this on Wikipedia's "evaluation strategy" page, but in short, I think of this as the function receiving a pointer to the object that's passed in.

This has a few repercussions. First, it means that there's no copy of the object. Under the covers, the function essentially receives a pointer that says, "You can find this `Person` instance at so-and-so memory address in the computer's RAM."

Second, if the object has mutable fields, the function can mutate those fields. When a function receives a `Person` instance and the `name` field is a `var`, the function can change the `name`:

```
def changeName(p: Person) = {
    p.name = "Al"
}
```

This change affects the `Person` instance that was passed in.

In regards to the name "by-value," the book, Programming Scala, makes this statement:

> "Typically, parameters to functions are by-value parameters; that is, the value of the parameter is determined before it is passed to the function."

In summary, in Scala the term "call by-value" means that the value is either:

- A primitive value (like an `Int`) that can't be changed

- A pointer to an object (like `Person`)

Background: By-name parameters

"By-name" parameters are quite different than by-value parameters. Rob Norris, (aka, "tpolecat") makes the observation that you can think about the two types of parameters like this:

- A *by-value* parameter is like receiving a `val` field; its body is evaluated once, when the parameter is bound to the function.
- A *by-name* parameter is like receiving a `def` method; its body is evaluated whenever it is used inside the function.

Those statements aren't 100% accurate, but they are decent analogies to start with.

A little more accurately, the book Scala Puzzlers says that by-name parameters are "evaluated only when they are referenced inside the function." The Scala Language Specification adds this:

> This (by-name) indicates that the argument is not evaluated at the point of function application, but instead is evaluated at each use within the function.

According to Wikipedia these terms date back to a language named ALGOL 60 (yes, the year 1960). But for me, the term "by-name" isn't very helpful. When you look at those quotes from the Puzzlers book and the Language Specification, you see that they both say, "a by-name parameter is only evaluated when it's accessed inside a function." Therefore, I find that the following names are more accurate and meaningful than "by-name":

- Call on access
- Evaluate on access
- Evaluate on use
- Evaluate when accessed

- Evaluate when referenced

However, because I can't change the universe, I'll continue to use the terms "by-name" and "call by-name" in this lesson, but I wanted to share those alternate names, which I think are more meaningful.

Example: Creating a timer

Okay, that's enough background about the names. Let's look at some code that shows how to create a by-name parameter, and what it gives you.

On Unix systems you can run a `time` command (`timex` on some systems) to see how long commands take to execute:

```
$ time find . -name "*.scala"
```

That command returns the results of the `find` command it was given, along with the time it took to run. The `time` portion of the output looks like this:

```
real  0m4.351s
user  0m0.491s
sys   0m1.341s
```

This is cool, and it can be a helpful way to troubleshoot performance problems. In fact, seeing how cool it is, you decide that you'd like to create a similar "timer" method in Scala.

Designing a Scala timer

Thinking in advance about how your new `timer` function should work, you decide that a nice API will let you write code like this:

```
val (result, time) = timer(someLongRunningAlgorithm)
```

and this:

```
val (result, time) = timer {
    ...
    ...
}
```

A `timer` like this gives you both the result of the algorithm and the time it took to run.

Assuming you already have a working `timer`, you can see how this works by running an example in the REPL:

```
scala> val (result, time) = timer{ Thread.sleep(500); 1 }
result: Int = 1
time: Double = 500.32
```

As expected, the code block returns the value `1`, with an execution time of about 500 ms.

Trying to define a function signature

Having just seen how to define signatures for function input parameters in the previous lessons, you realize that you know how to write a `timer` … or at least you think you can.

The problem you run into right away is, "Just what is that algorithm that's being passed in?" It could look like this:

```
def timer(f:(Int) => Int) ...
```

or this:

```
def timer(f:(Double) => Double) ...
```

or anything else:

```
def timer(f:() => Unit) ...
def timer(f:(Person) => String) ...
def timer(f:(Pizza, Order) => Double) ...
def timer(f:(Pizza, Order, Customer, Discounts) => Currency) ...
```

"Hmm," you begin thinking, "this is quite a problem …"

Fortunately the Scala creators gave us a nice solution for problems like these.

By-name syntax

The solution for situations like this is to use Scala's by-name syntax. It's similar to defining function input parameters, but it also makes problems like this simple.

The general syntax for defining a by-name parameter looks like this:

```
def timer(blockOfCode: => theReturnType) ...
```

If you look back at the function input parameter examples, you'll see that the by-name syntax is similar to this example:

```
def timer(f:() => Unit) ...
```

The main difference is that you leave off the `()` after the input parameter.

Given that brief introduction to the by-name syntax, to create a `timer` that can accept a block of code that returns any type, you make the return type generic. Therefore, I can sketch the `timer` signature like this:

```
def timer[A](blockOfCode: => A) = ???
```

With that signature in hand, I can then complete the `timer` function like this:

```
def timer[A](blockOfCode: => A) = {
    val startTime = System.nanoTime
    val result = blockOfCode
    val stopTime = System.nanoTime
    val delta = stopTime - startTime
    (result, delta/1000000d)
}
```

The `timer` method uses the by-name syntax to accept a block of code as an input parameter. Inside the `timer` function there are three lines of code that deal with determining how long the `blockOfCode` takes to run, with this line sandwiched in between those time-related expressions:

```
    val result = blockOfCode
```

That line (a) executes `blockOfCode` and (b) assigns its return value to `result`. Because `blockOfCode` is defined to return a generic type (`A`), it may return `Unit`, an `Int`, a `Double`, a `Seq[Person]`, a `Map[Person, Seq[Person]]`, whatever.

Now you can use the `timer` function for all sorts of things. It can be used for something that isn't terribly useful, like this:

```
scala> val (result, time) = timer(println("Hello"))
Hello
result: Unit = ()
time: Double = 0.160
```

It can be used for an algorithm that reads a file and returns an iterator:

```
scala> def readFile(filename: String) = io.Source.fromFile(filename).getLines
readFile: (filename: String)Iterator[String]

scala> val (result, time) = timer(readFile("/etc/passwd"))
result: Iterator[String] = non-empty iterator
time: Double = 32.119
```

Or it can be used for just about anything else:

```
val (result, time) = timer{ someLongRunningAlgorithmThatReturnsSomething }
```

"When is my code block run?"

A great question right now is, "When are my by-name parameters executed?"

In the case of the `timer` function, it executes the `blockOfCode` when the second line of the function is reached. But if that doesn't satisfy your curious mind, you can create another example like this:

```
def test[A](codeBlock: => A) = {
    println("before 1st codeBlock")
    val a = codeBlock
    println(a)
    Thread.sleep(10)

    println("before 2nd codeBlock")
    val b = codeBlock
    println(b)
    Thread.sleep(10)

    println("before 3rd codeBlock")
    val c = codeBlock
    println(c)
}
```

If you paste that code into the Scala REPL, you can then test it like this:

```
scala> test( System.currentTimeMillis )
```

That line of code will produce output like this:

```
before 1st codeBlock
1480206447942
before 2nd codeBlock
1480206447954
before 3rd codeBlock
1480206447966
```

As that output shows, the block of code that is passed in is executed each time it's referenced inside the function.

Another example: A Swing utility

As another example of how I use this technique, when I was writing a lot of Swing (GUI) code with Scala, I wrote this `invokeLater` function to accept blocks of code that should be run on the JVM's Event Dispatch Thread (EDT):

```
def invokeLater(codeBlock: => Unit) {
    SwingUtilities.invokeLater(new Runnable() {
        def run() {
            codeBlock
        }
    });
}
```

`invokeLater` defines `codeBlock` as a by-name input parameter, and `codeBlock` is expected to return `Unit` (nothing). I defined it like that because every block of code it accepts is intended to update the Swing GUI, which means that each code block is used to achieve that side effect.

As an example, here are two example calls to `invokeLater` from my Sarah application:

```
invokeLater(mainFrame.setSarahIsSleeping())
invokeLater(mainFrame.setSarahIsListening())
```

In these examples, `mainFrame.setSarahIsSleeping()` and `mainFrame.setSarahIsListening()` are both function calls, and those functions do whatever they need to do to update the Sarah's Swing GUI.

Knowing how those functions work, if for some reason I didn't have them written as functions, I could have passed this block of code into `invokeLater` to achieve the same effect as the first example:

```
invokeLater {
    val controller = mainController.getMainFrameController()
    controller.setBackground(SARAH_IS_SLEEPING_COLOR)
}
```

Either approach — passing in a function, or passing in a block of code — is valid.

Why have by-name parameters?

Programming in Scala, written by Martin Odersky and Bill Venners, provides a great example of why by-name parameters were added to Scala. Their example goes like this:

1. Imagine that Scala does not have an `assert` function, and you want one.
2. You attempt to write one using function input parameters, like this:

```
def myAssert(predicate: () => Boolean) =
    if (assertionsEnabled && !predicate())
        throw new AssertionError
```

That code uses the "function input parameter" techniques I showed in previous lessons, and assuming that the variable `assertionsEnabled` is in scope, it will compile just fine.

The problem is that when you go to use it, you have to write code like this:

```
myAssert(() => 5 > 3)
```

Because `myAssert` states that `predicate` is a function that takes no input parameters and returns a `Boolean`, that's how you have to write this line of code. It works, but it's not pleasing to the eye.

The solution is to change `predicate` to be a by-name parameter:

```
def byNameAssert(predicate: => Boolean) =
    if (assertionsEnabled && !predicate)
        throw new AssertionError
```

With that simple change, you can now write assertions like this:

```
byNameAssert(5 > 3)
```

That's much more pleasing to look at than this:

```
myAssert(() => 5 > 3)
```

Programming in Scala states that this is the primary use case for by-name parameters:

> The result is that using `byNameAssert` looks exactly like using a built-in control structure.

If you want to experiment with this code, here's the source code for a small but complete test class I created from their example:

```
object ByNameTests extends App {

    var assertionsEnabled = true

    def myAssert(p: () => Boolean) =
        if (assertionsEnabled && !p())
            throw new AssertionError

    myAssert(() => 5 > 3)

    def byNameAssert(p: => Boolean) =
```

```
        if (assertionsEnabled && !p)
            throw new AssertionError

    byNameAssert(5 > 3)

}
```

As you can see from that code, there's only a small syntactical difference between defining a function input parameter that takes no input parameters and a by-name parameter:

```
p: () => Boolean    // a function input parameter
p: => Boolean       // a by-name parameter
```

As you can also tell from these two lines:

```
myAssert(() => 5 > 3)
byNameAssert(5 > 3)
```

you need to call them differently.

Summary

This lesson showed:

- The differences between by-value and by-name parameters
- Examples of the by-name syntax
- How to use by-name parameters in your functions
- Examples of when by-name parameters are appropriate
- Some comparisons of by-name parameters and higher-order functions

See also

- On StackOverflow, Daniel Sobral has a nice answer to a question about the difference between `(f: A => B)` and `(f: () => A)`
- Scala Puzzlers comments about function input parameters
- Evaluation strategy on Wikipedia

27

Functions Can Have Multiple Parameter Groups

> *“Logic clearly dictates that the needs of the many outweigh the needs of the few.”*
>
> Spock in *Star Trek II: The Wrath of Khan*

Introduction

Scala lets you create functions that have multiple input parameter groups, like this:

```
def foo(a: Int, b: String)(c: Double)
```

Because I knew very little about FP when I first started working with Scala, I originally thought this was just some sort of syntactic nicety. But then I learned that one cool thing this does is that it enables you to write your own control structures. For instance, you can write your own `while` loop, and I show how to do that in this lesson.

Beyond that, the book Scala Puzzlers states that being able to declare multiple parameter groups gives you these additional benefits (some of which are advanced and I rarely use):

- They let you have both implicit and non-implicit parameters
- They facilitate type inference
- A parameter in one group can use a parameter from a previous group as a default value

I demonstrate each of these features in this lesson, and show how multiple parameter groups are used to create partially-applied functions in the next lesson.

Goals

The goals of this lesson are:

- Show how to write and use functions that have multiple input parameter groups
- Demonstrate how this helps you create your own control structures, which in turn can help you write your own DSLs
- Show some other potential benefits of using multiple input parameter groups

First example

Writing functions with multiple parameter groups is simple. Instead of writing a "normal" `add` function with one parameter group like this:

```
def add(a: Int, b: Int, c: Int) = a + b + c
```

just put your function's input parameters in different groups, with each group surrounded by parentheses:

```
def sum(a: Int)(b: Int)(c: Int) = a + b + c
```

After that, you can call `sum` like this:

```
scala> sum(1)(2)(3)
res0: Int = 6
```

That's all there is to the basic technique. The rest of this lesson shows the advantages that come from using this approach.

A few notes about this technique

Note that when you write `sum` with three input parameter groups like this, trying to call it with three parameters in one group won't work:

```
scala> sum(1,2,3)
<console>:12: error: too many arguments for method
sum: (a: Int)(b: Int)(c: Int)Int
       sum(1,2,3)
          ^
```

You must supply the input parameters in three separate input lists.

Another thing to note is that each parameter group can have multiple input parameters:

```
def doFoo(firstName: String, lastName: String)(age: Int) = ???
```

How to write your own control structures

To show the kind of things you can do with multiple parameter groups, let's build a control structure of our own. To do this, imagine for a moment that you don't like the built-in Scala `while` loop — or maybe you want to add some functionality to it — so you want to create your own `whilst` loop, which you can use like this:

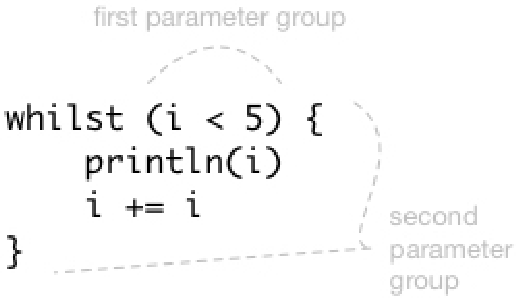
```
var i = 0
whilst (i < 5) {
    println(i)
    i += 1
}
```

> Note: I use a `var` field here because I haven't covered recursion yet.

A thing that your eyes will soon learn to see when looking at code like this is that `whilst` *must* be defined to have two parameter groups. The first parameter group is `i`

`< 5`, which is the expression between the two parentheses. Note that this expression yields a `Boolean` value. Therefore, by looking at this code you know `whilst` must be defined so that it's first parameter group is expecting a `Boolean` parameter of some sort.

The second parameter group is the block of code enclosed in curly braces immediately after that. These two groups are highlighted in Figure 27.1.



```
whilst (i < 5) {
    println(i)
    i += i
}
```

Figure 27.1: The second parameter group is enclosed in the curly braces

You'll see this pattern *a lot* in Scala/FP code, so it helps to get used to it.

I demonstrate more examples in this chapter, but the lesson for the moment is that when you see code like this, you should think:

- I see a function named `whilst` that has two parameter groups
- The first parameter group must evaluate to a `Boolean` value
- The second parameter group appears to return nothing (`Unit`), because the last expression in the code block (`i += 1`) returns nothing

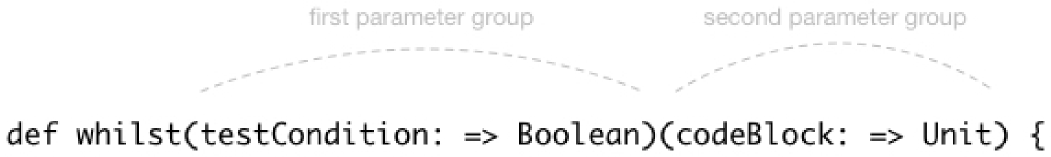
How to create `whilst`

To create the `whilst` control structure, define it as a function that takes two parameter groups. As mentioned, the first parameter group must evaluate to a `Boolean` value, and the second group takes a block of code that evaluates to `Unit`; the user wants to run this block of code in a loop as long as the first parameter group evaluates to `true`.

When I write functions these days, the first thing I like to do is sketch the function's signature, and the previous paragraph tells me that `whilst`'s signature should look like this:

```
def whilst(testCondition: => Boolean)(codeBlock: => Unit) = ???
```

The two parameters groups are highlighted in Figure 27.2.



```
def whilst(testCondition: => Boolean)(codeBlock: => Unit) {
```

Figure 27.2: The two parameter groups in `whilst`*'s function signature*

Using by-name parameters

Notice that both parameter groups use *by-name* parameters. The first parameter (`testCondition`) must be a by-name parameter because it specifies a test condition that will repeatedly be tested inside the function. If this *wasn't* a by-name parameter, the `i < 5` code shown here:

```
var i = 0
whilst (i < 5) ...
```

would immediately be translated by the compiler into this:

```
whilst (0 < 5) ...
```

and then that code would be further "optimized" into this:

```
whilst (true) ...
```

If this happens, the `whilst` function would receive `true` for its first parameter, and the loop will run forever. This would be bad.

But when `testCondition` is defined as a by-name parameter, the `i < 5` test condition code block is passed into `whilst` without being evaluated, which is what we desire.

Using a by-name parameter in the *last* parameter group when creating control structures is a common pattern in Scala/FP. This is because as I just showed, a by-name parameter lets the consumer of your control structure pass in a block of code to solve their problem, typically enclosed in curly braces, like this:

```
customControlStructure(...) {
   // custom code block here
   ...
   ...
}
```

The final code

So far, I showed that the `whilst` signature begins like this:

```
def whilst(testCondition: => Boolean)(codeBlock: => Unit) = ???
```

In FP, the proper way to implement `whilst`'s body is with recursion, but because I haven't covered that yet, I'm going to cheat here and implement `whilst` with an inner `while` loop. Admittedly that's some serious cheating, but for the purposes of this lesson I'm not really interested in the body of `whilst`; I'm interested in its signature, along with what this general approach lets you accomplish.

Therefore, having defined `whilst`'s signature, this is what `whilst` looks like as a wrapper around a `while` loop:

```
def whilst(testCondition: => Boolean)(codeBlock: => Unit) {
    while (testCondition) {
        codeBlock
    }
}
```

Note that `whilst` doesn't return anything. That's *implied* by the current function signature, and you can make it more explicit by adding a `Unit` return type to the function signature:

```
def whilst(testCondition: => Boolean)(codeBlock: => Unit): Unit = {
                                                         --------
```

With that change, the final `whilst` function looks like this:

```
def whilst(testCondition: => Boolean)(codeBlock: => Unit): Unit = {
    while (testCondition) {
        codeBlock
    }
}
```

Using `whilst`

Because I cheated with the function body, that's all there is to writing `whilst`. Now you can use it anywhere you would use `while`. This is one possible example:

```
var i = 1
whilst(i < 5) {
    println(i)
    i += 1
}
```

Exercise: Write a control structure using three parameter groups

The `whilst` example shows how to write a custom control structure using two parameter groups. It also shows a common pattern:

- Use one or more parameter groups to break the input parameters into different “compartments”
- Specifically define the parameter in the last parameter group as a by-name parameter so the function can accept a custom block of code

Control structures can have more than two parameter lists. As an exercise, imagine that you want to create a control structure that makes it easy to execute a condition if two test conditions are both true. Imagine the control structure is named `ifBothTrue`, and it will be used like this:

```
ifBothTrue(age > 18)(numAccidents == 0) {
    println("Discount!")
}
```

Just by looking at that code, you should be able to answer these questions:

- How many input parameter groups does `ifBothTrue` have?
- What is the type of the first group?
- What is the type of the second group?
- What is the type of the third group?

Sketch the signature of the `ifBothTrue` function. Start by sketching *only* the function signature, as I did with the `whilst` example:

Once you're confident that you have the correct function signature, sketch the function body here:

Solution

In this case, because `ifBothTrue` takes two test conditions followed by a block of code, and it doesn't return anything, its signature looks like this:

```
def ifBothTrue(test1: => Boolean)(test2: => Boolean)(codeBlock: => Unit): Unit = ???
```

Because the code block should only be run if both test conditions are `true`, the complete function should be written like this:

```
def ifBothTrue(test1: => Boolean)(test2: => Boolean)(codeBlock: => Unit): Unit = {
    if (test1 && test2) {
        codeBlock
    }
}
```

You can test `ifBothTrue` with code like this:

```
val age = 19
val numAccidents = 0
ifBothTrue(age > 18)(numAccidents == 0) { println("Discount!") }
```

This also works:

```
ifBothTrue(2 > 1)(3 > 2)(println("hello"))
```

A favorite control structure

One of my favorite uses of this technique is described in the book, Beginning Scala. In that book, David Pollak creates a `using` control structure that automatically calls the `close` method on an object you give it. Because it automatically calls `close` on the object you supply, a good example is using it with a database connection.

The `using` control structure lets you write clean database code like the following example, where the database connection `conn` is automatically close after the `save` call:

```
def saveStock(stock: Stock) {
    using(MongoFactory.getConnection()) { conn =>
        MongoFactory.getCollection(conn).save(buildMongoDbObject(stock))
    }
}
```

In this example the variable `conn` comes from the `MongoFactory.getConnection()` method. `conn` is an instance of a `MongoConnection`, and the `MongoConnection` class defines `close` method, which is called automatically by `using`. (If `MongoConnection` did not have a `close` method, this code would not work.)

If you want to see how `using` is implemented, I describe it in my article, Using the `using` control structure from Beginning Scala

Benefit: Using implicit values

A nice benefit of multiple input parameter groups comes when you use them with *implicit* parameters. This can help to simplify code when a resource is needed, but passing that resource explicitly to a function makes the code harder to read.

To demonstrate how this works, here's a function that uses multiple input parameter groups:

```
def printIntIfTrue(a: Int)(implicit b: Boolean) = if (b) println(a)
```

Notice that the `Boolean` in the second parameter group is tagged as an `implicit` value, but don't worry about that just yet. For the moment, just note that if you paste this function into the REPL and then call it with an `Int` and a `Boolean`, it does what it looks like it should do, printing the `Int` when the `Boolean` is `true`:

```
scala> printIntIfTrue(42)(true)
42
```

Given that background, let's see what that `implicit` keyword on the second parameter does for us.

Using implicit values

Because `b` is defined as an *implicit* value in the *last* parameter group, if there is an implicit `Boolean` value in scope when `printIntIfTrue` is invoked, `printIntIfTrue` can use that `Boolean` without you having to explicitly provide it.

You can see how this works in the REPL. First, as an intentional error, try to call `printIntIfTrue` without a second parameter:

```
scala> printIntIfTrue(1)
<console>:12: error: could not find implicit value for parameter b: Boolean
       printIntIfTrue(1)
                     ^
```

Of course that fails because `printIntIfTrue` requires a `Boolean` value in its second parameter group. Next, let's see what happens if we define a regular `Boolean` in the current scope:

```
scala> val boo = true
boo: Boolean = true

scala> printIntIfTrue(1)
<console>:12: error: could not find implicit value for parameter b: Boolean
       printIntIfTrue(1)
```

```
                     ^
```

Calling `printIntIfTrue` still fails, and the reason it fails is because there are no *implicit* `Boolean` values in scope when it's called. Now note what happens when `boo` is defined as an implicit `Boolean` value and `printIntIfTrue` is called:

```
scala> implicit val boo = true
boo: Boolean = true

scala> printIntIfTrue(33)
33
```

`printIntIfTrue` works with only one parameter!

This works because:

1. The `Boolean` parameter in `printIntIfTrue`'s last parameter group is tagged with the `implicit` keyword
2. `boo` is declared to be an implicit `Boolean` value

The way this works is like this:

1. The Scala compiler knows that `printIntIfTrue` is defined to have two parameter groups.
2. It also knows that the second parameter group declares an implicit `Boolean` parameter.
3. When `printIntIfTrue(33)` is called, only one parameter group is supplied.
4. At this point Scala knows that one of two things must now be true. Either (a) there better be an implicit `Boolean` value in the current scope, in which case Scala will use it as the second parameter, or (b) Scala will throw a compiler error.

Because `boo` is an implicit `Boolean` value and it's in the current scope, the Scala compiler reaches out and automatically uses it as the input parameter for the second parameter group. That is, `boo` is used just as though it had been passed in explicitly.

The benefit

If that code looks too "magical," I'll say two things about this technique:

- It works really well in certain situations
- Don't overuse it, because when it's used wrongly it makes code hard to understand and maintain (which is pretty much an anti-pattern)

An area where this technique works really well is when you need to refer to a shared resource several times, and you want to keep your code clean. For instance, if you need to reference a database connection several times in your code, using an implicit connection can clean up your code. It tends to be obvious that an implicit connection is hanging around, and of course database access code isn't going to work without a connection.

An implicit execution context

A similar example is when you need an "execution context" in scope when you're writing multi-threaded code with the Akka library. For example, with Akka you can create an implicit `ActorSystem` like this early in your code:

```
implicit val actorSystem = ActorSystem("FutureSystem")
```

Then, at one or more places later in your code you can create a `Future` like this, and the `Future` "just works":

```
val future = Future {
    1 + 1
}
```

The reason this `Future` works is because it is written to look for an implicit `ExecutionContext`. If you dig through the Akka source code you'll see that `Future`'s `apply` method is written like this:

```
def apply [T] (body: => T)(implicit executor: ExecutionContext) ...
```

As that shows, the `executor` parameter in the last parameter group is an `implicit` value of the `ExecutionContext` type. Because an `ActorSystem` is an instance of an `ExecutionContext`, when you define the `ActorSystem` as being `implicit`, like this:

```
implicit val actorSystem = ActorSystem("FutureSystem")
--------
```

`Future`'s `apply` method can find it and "pull it in" automatically. This makes the `Future` code much more readable. If `Future` didn't use an `implicit` value, each invocation of a new `Future` would have to look something like this:

```
val future = Future(actorSystem) {
    code to run here ...
}
```

That's not too bad with just one `Future`, but more complicated code is definitely cleaner without it repeatedly referencing the `actorSystem`.

> If you're new to Akka Actors, my article, A simple working Akka Futures example, explains everything I just wrote about actors, futures, execution contexts, and actor systems.

> If you know what an `ExecutionContext` is, but don't know what an `ActorSystem` is, it may help to know that you can also use an `ExecutionContext` as the implicit value in this example. So instead of using the `ActorSystem` as shown in the example, just create an implicit `ExecutionContext`, like this:
>
> ```
> val pool = Executors.newCachedThreadPool()
> implicit val ec = ExecutionContext.fromExecutorService(pool)
> ```
>
> After that you can create a `Future` as before:

```
val future = Future { 1 + 1 }
```

Limits on implicit parameters

The Scala language specification tells us these things about implicit parameters:

- A method or constructor can have only one implicit parameter list, and it must be the last parameter list given
- If there are several eligible arguments which match the implicit parameter's type, a most specific one will be chosen using the rules of static overloading resolution

I'll show some of what this means in the following "implicit parameter FAQs".

FAQ: Can you use `implicit` more than once in your parameter lists?

No, you can't. This code will not compile:

```
def printIntIfTrue(implicit a: Int)(implicit b: Boolean) = if (b) println(a)
```

The REPL shows the error message you'll get:

```
scala> def printIntIfTrue(implicit a: Int)(implicit b: Boolean) = if (b) println(a)
<console>:1: error: '=' expected but '(' found.
def printIntIfTrue(implicit a: Int)(implicit b: Boolean) = if (b) println(a)
                                   ^
```

FAQ: Does the `implicit` have to be in the last parameter list?

Yes. This code, with an `implicit` in the first list, won't compile:

```
def printIntIfTrue(implicit b: Boolean)(a: Int) = if (b) println(a)
```

The REPL shows the compiler error:

```
scala> def printIntIfTrue(implicit b: Boolean)(a: Int) = if (b) println(a)
<console>:1: error: '=' expected but '(' found.
def printIntIfTrue(implicit b: Boolean)(a: Int) = if (b) println(a)
                                       ^
```

FAQ: What happens when multiple implicit values are in scope and can match the parameter?

In theory, as the Specification states, “a most specific one will be chosen using the rules of static overloading resolution.” In practice, if you find that you’re getting anywhere near this situation, I wouldn’t use implicit parameters.

A simple way to show how this fails is with this series of expressions:

```
def printIntIfTrue(a: Int)(implicit b: Boolean) = if (b) println(a)
implicit val x = true
implicit val y = false
printIntIfTrue(42)
```

When you get to that last expression, can you guess what will happen?

What happens is that the compiler has no idea which `Boolean` should be used as the implicit parameter, so it bails out with this error message:

```
scala> printIntIfTrue(42)
<console>:14: error: ambiguous implicit values:
 both value x of type => Boolean
 and value y of type => Boolean
 match expected type Boolean
       printIntIfTrue(42)
```

```
                    ^
```

This is a simple example of how using implicit parameters can create a problem.

A more complicated example

If you want to see a more complicated example of how implicit parameters can create a problem, read this section. Otherwise, feel free to skip to the next section.

Here's another example that should provide fair warning about using this technique. Given (a) the following trait and classes:

```
trait Animal
class Person(name: String) extends Animal {
    override def toString = "Person"
}
class Employee(name: String) extends Person(name) {
    override def toString = "Employee"
}
```

(b) define a method that uses an implicit `Person` parameter:

```
// uses an `implicit` Person value
def printPerson(b: Boolean)(implicit p: Person) = if (b) println(p)
```

and then (c) create implicit instances of a `Person` and an `Employee`:

```
implicit val p = new Person("person")
implicit val e = new Employee("employee")
```

Given that setup, and knowing that "a most specific one (implicit instance) will be chosen using the rules of static overloading resolution," what would you expect this statement to print?:

```
printPerson(true)
```

If you guessed `Employee`, pat yourself on the back:

```
scala> printPerson(true)
Employee
```

(I didn't guess `Employee`.)

If you know the rules of "static overloading resolution" better than I do, what do you think will happen if you add this code to the existing scope:

```
class Employer(name: String) extends Person(name) {
    override def toString = "Employer"
}
implicit val r = new Employer("employer")
```

and then try this again:

```
printPerson(true)
```

If you said that the compiler would refuse to participate in this situation, you are correct:

```
scala> printPerson(true)
<console>:19: error: ambiguous implicit values:
 both value e of type => Employee
 and value r of type => Employer
 match expected type Person
       printPerson(true)
                  ^
```

As a summary, I think this technique works great when there's only one implicit value in scope that can possibly match the implicit parameter. If you try to use this with multiple implicit parameters in scope, you really need to understand the rules of application. (And I further suggest that once you get away from your code for a while, you'll eventually forget those rules, and the code will be hard to maintain. This is nobody's goal).

Using default values

As the Scala Puzzlers book notes, you can supply default values for input parameters when using multiple parameter groups, in a manner similar to using one parameter group. Here I specify default values for the parameters `a` and `b`:

```
scala> def f2(a: Int = 1)(b: Int = 2) = { a + b }
f2: (a: Int)(b: Int)Int
```

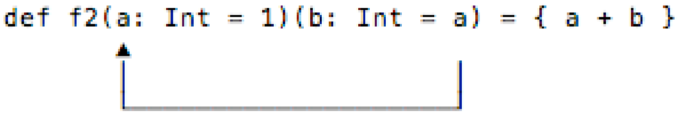
That part is easy, but the "magic" in this recipe is knowing that you need to supply empty parentheses when you want to use the default values:

```
scala> f2()()
res0: Int = 3

scala> f2(10)()
res1: Int = 12

scala> f2()(10)
res2: Int = 11
```

As the Puzzlers book also notes, a parameter in the second parameter group can use a parameter from the first parameter group as a default value. In this next example I assign `a` to be the default value for the parameter `b`:

```
def f2(a: Int = 1)(b: Int = a) = { a + b }
```

Figure 27.3 makes this more clear.



```
def f2(a: Int = 1)(b: Int = a) = { a + b }
```

Figure 27.3: `a` in the second parameter group is the same `a` in the first parameter group

The REPL shows that this works as expected:

```
scala> def f2(a: Int = 1)(b: Int = a) = { a + b }
f2: (a: Int)(b: Int)Int

scala> f2()()
res0: Int = 2
```

I haven't had a need for these techniques yet, but in case you ever need them, there you go.

Summary

In this lesson I covered the following:

- I showed how to write functions that have multiple input parameter groups.
- I showed how to call functions that have multiple input parameter groups.
- I showed to write your own control structures, such as `whilst` and `ifBothTrue`. The keys to this are (a) using multiple parameter groups and (b) accepting a block of code as a by-name parameter in the last parameter group.
- I showed how to use `implicit` parameters, and possible pitfalls of using them.
- I showed how to use default values with multiple parameter groups.

What's next

The next lesson expands on this lesson by showing what "Currying" is, and by showing how multiple parameter groups work with partially-applied functions.

See Also

- My article, Using the `using` control structure from Beginning Scala
- Joshua Suereth's scala-arm project is similar to the `using` control structure
- The Scala "Breaks" control structure is created using the techniques shown in

this lesson, and I describe it in my article, How to use break and continue in Scala

28

Partially-Applied Functions (and Currying)

Motivation

My motivations for writing this lesson are a little different than usual. Typically I think, “You’ll want to know this feature so you can use it like ___,” but the first motivation for this lesson goes like this: You’ll want to know about the concept of “currying” because experienced FP developers talk about it a lot, especially if they have Haskell programming experience. (I did mention that Haskell was named after Haskell *Curry*, didn’t I?)

A second motivation is that the concept of currying is related to the multiple parameter groups I showed in the previous lesson come from.

The primary motivation for writing this lesson is that having multiple parameter groups make it a little easier to create *partially-applied functions*, and these can be useful in your FP code.

I’ll cover all of these topics in this lesson.

Goals

Given that introduction, the goals of this lesson are:

- Provide a definition of *currying*
- Show how to create partially-applied functions from functions that have (a) multiple parameter groups or (b) single parameter groups

I’ll also show how to create “curried” functions from regular functions, and show how Scala gets these features to work with the JVM.

Currying

When I first got started in FP, I got lost in some of the nomenclature, and “currying” was a particularly deep rabbit’s hole of “Time in My Life I Wish I Had Spent Differently.”

All that the theory of *currying* means is that a function that takes *multiple* arguments can be translated into a series of function calls that each take a *single* argument. In pseudocode, this means that an expression like this:

```
result = f(x)(y)(z)
```

is mathematically the same as something like this:

```
f1 = f(x)
f2 = f1(y)
result = f2(z)
```

That’s all it means. The Wikipedia page on Currying describes the theory of currying like this:

> In mathematics and computer science, currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.

They later state:

> There are analytical techniques that can only be applied to functions with a single argument. Practical functions frequently take more arguments than this.

What this means

In my daily working life, this sort of theory usually isn’t important. It’s one of those things that’s “nice to know,” but the important things are really (a) how this impacted the design of the Scala language, and (b) what you can do because of this theory.

In Scala this seems to fit most naturally with functions that have multiple input parameters groups, and I'll demonstrate that in this lesson.

A terminology note

In the remainder of this lesson I'll occasionally use the acronym "PAF" to mean "partially-applied function."

Partially-applied functions

To understand PAFs, I'll start with two definitions from this online JavaScript course:

1) *Application*: The process of applying a function to its arguments in order to produce a return value.

   As in algebra, in FP you say that "a function is applied to its arguments," so "Application" in this context can also be called "Full Application," or "Complete Application."

2) *Partial Application*: This is the process of applying a function to *some* of its arguments. A partially-applied function gets returned for later use. In other words, a PAF is a function that takes a function with multiple parameters and returns a function with fewer parameters.

The best way to explain PAFs is with examples, so let's look at a few.

Example 1 (partially-applied functions)

The following example shows how PAFs work. In the first step, you define a function with multiple parameter groups:

```
scala> def plus(a: Int)(b: Int) = a + b
plus: (a: Int)(b: Int)Int
```

Next, rather than giving the function all of the parameters in the two parameter groups it specifies, you give it (a) the parameter for the first group (`a`), and (b) a placeholder for the parameter in the second list, the ubiquitous underscore character:

```
scala> def plus2 = plus(2)(_)
plus2: Int => Int
```

The REPL output shows that this creates a new function named `plus2` which has the type `Int => Int`. This means that `plus2` takes an `Int` as input, and returns an `Int` as a result.

At this point you can think of `plus2` as looking like this:

```
def plus(b: Int) = 2 + b
```

`plus2` has been "seeded" with the initial `Int` value `2`, and now it's just sitting there, waiting for another `Int` value that it can add to it. Let's give it another `2`:

```
scala> plus2(2)
res0: Int = 4
```

Here's what it looks like when you give it a `3`:

```
scala> plus2(3)
res1: Int = 5
```

As this shows, `plus2` gladly adds `2` to any `Int` it is given.

Before I move on to another example, note that you can create `plus2` in either of these ways:

```
def plus2 = plus(2)(_)
def plus2 = plus(2)_
```

I prefer the first approach, but some people prefer the second approach.

Example 2 (partially-applied functions)

The general benefit that this approach gives you is that it's a way to create specialized methods from more general methods. I demonstrate that in the Scala Cookbook, and I'll share a variation of that example here.

When you're emitting HTML from Scala code, a `wrap` function that adds a prefix and a suffix to an HTML snippet can be really useful:

```
def wrap(prefix: String)(html: String)(suffix: String) = {
    prefix + html + suffix
}
```

You can use that function to do something like this, where I wrap a string in opening and closing `<div>` tags:

```
val hello = "Hello, world"
val result = wrap("<div>")(hello)("</div>")
```

Of course that `<div>` tag can be more complicated, such as specifying a CSS `class` or `id`, but I'm keeping this simple.

It turns out that `wrap` is a really nice, *general* function, so you can wrap text in `DIV` tags, `P` tags, `SPAN` tags, etc. But if you're going to be wrapping a lot of strings with `DIV` tags, what you probably want is a more specific `wrapWithDiv` function. This is a great time to use a partially-applied function, because that's what they do, helping you create a *specific* function from a *general* function:

```
val wrapWithDiv = wrap("<div>")(_: String)("</div>")
```

Now you can call `wrapWithDiv`, just passing it the HTML you want to wrap:

```
scala> wrapWithDiv("<p>Hello, world</p>")
res0: String = <div><p>Hello, world</p></div>
```

```
scala> wrapWithDiv("<img src=\"/images/foo.png\" />")
res1: String = <div><img src="/images/foo.png" /></div>
```

As a nice benefit, you can still call the original `wrap` function:

```
wrap("<pre>", "val x = 1", "</pre>")
```

and you can also create other, more-specific functions:

```
val wrapWithPre = wrap("<pre>")(_: String)("</pre>")
```

It's worth noting that you make a more specific function by "seeding" the more general function with one or more initial parameters. That is, you partially-apply parameters to the general function to make the specific function.

Handling the missing parameter

It's necessary to specify the *type* of the missing parameter, as I did in this code:

```
val wrapWithDiv = wrap("<div>")(_: String)("</div>")
                                ---------
```

If you don't specify the type, you'll get a compiler error that looks like this:

```
scala> val wrapWithDiv = wrap("<div>")(_)("</div>")
<console>:11: error: missing parameter type for
expanded function ((x$1) => wrap("<div>")(x$1)("</div>"))
       val wrapWithDiv = wrap("<div>")(_)("</div>")
                                       ^
```

Summary: Partially-applied functions

As a summary, PAFs give you this capability:

- You write a general function
- You create a specific function from the general function
- You still have access to both functions, and you kept your code "DRY" — you didn't copy and paste code to make the new function

Creating curried functions from regular functions

As a fun example of some things you can do with PAFs, the "partially-applied functions" section of the Scala Exercises website demonstrates that you can create curried functions from "normal" Scala functions. For instance, you can start with a "normal" one-parameter group function like this:

```
def add(x: Int, y: Int) = x + y
```

Then they show that you can create a `Function2` instance from `add` by adding an underscore after it, like this:

```
scala> val addFunction = add _
addFunction: (Int, Int) => Int = <function2>
```

They then prove that it's a `Function2` instance like this:

```
(add _).isInstanceOf[Function2[_, _, _]]
```

> This technique of converting a `def` method into a true function uses a Scala technology known as "Eta Expansion." I mentioned this in the previous lessons, and I also discuss it in depth in the appendix titled, "The Differences Between 'def' and 'val' When Defining Functions."

Then they create a "curried" function from that `Function2` instance:

```
val addCurried = (add _).curried
```

Now you can use the new curried function like this:

```
addCurried(1)(2)
```

As this shows, calling the `curried` method on the `add` function instance creates a new function that has two parameter groups. (So, a curried function can be thought of as a function with multiple parameter groups.)

It's also easy to create a partially-applied function from the curried function, like this:

```
val addCurriedTwo = addCurried(2)    // create a PAF
addCurriedTwo(10)                    // use the PAF
```

See it in the REPL

You can see how all of those steps work by pasting the code into the REPL:

```
scala> def add(x: Int, y: Int) = x + y
add: (x: Int, y: Int)Int

scala> (add _).isInstanceOf[Function2[_, _, _]]
res0: Boolean = true

scala> val addCurried = (add _).curried
addCurried: Int => (Int => Int) = <function1>

scala> addCurried(1)(2)
res1: Int = 3

scala> val addCurriedTwo = addCurried(2)
addCurriedTwo: Int => Int = <function1>

scala> addCurriedTwo(10)
res2: Int = 12
```

Personally, I mostly use curried functions to create control structures — as I demonstrated with `whilst` and `ifBothTrue` in the previous lesson. So, at the moment, this is a technique I know about, but have not used.

Partially-applied functions without multiple parameter groups

So far I've shown that you can create a partially-applied function with functions that have multiple parameter groups, but because Scala is really convenient, you can create PAFs with single parameter group functions as well.

To do this, first define a function as usual, with one parameter group:

```
def wrap(prefix: String, html: String, suffix: String) = {
    prefix + html + suffix
}
```

Then create a PAF by applying the first and third parameters, but not the second:

```
val wrapWithDiv = wrap("<div>", _: String, "</div>")
```

The `wrapWithDiv` function you create in this manner works the same as the `wrapWithDiv` function created in the previous example:

```
scala> val wrapWithDiv = wrap("<div>", _: String, "</div>")
wrapWithDiv: String => String = <function1>

scala> wrapWithDiv("Hello, world")
res1: String = <div>Hello, world</div>
```

Extra credit: How can all of this work with the JVM?

If you're interested in how things work under the covers, a good question at this point is, "How can this stuff possibly work with the JVM?" The JVM certainly wasn't written to account for things like currying and PAFs, so how does any of this work?

A short answer is that (a) the Scala compiler "uncurries" your code, and (b) you can see this during the compilation process. For example, write a little Scala class like this:

```
class Currying {
    def f1(a: Int, b: Int) = { a + b }   // 1 param group
    def f2(a: Int)(b: Int) = { a + b }   // 2 param groups
}
```

Then compile that class with this command:

```
$ scalac -Xprint:all Currying.scala
```

if you dig through the end of that output, you'll see that the Scala compiler has an "uncurry" phase. A short version of the tail end of the compiler output looks like this:

```
[[syntax trees at end of typer]] // Currying.scala
package <empty> {
  class Currying extends scala.AnyRef {
    def <init>(): Currying = {
      Currying.super.<init>();
      ()
    };
    def f1(a: Int, b: Int): Int = a.+(b);
    def f2(a: Int)(b: Int): Int = a.+(b)
  }
}

.
.
.

[[syntax trees at end of uncurry]] // Currying.scala
package <empty> {
  class Currying extends Object {
    def <init>(): Currying = {
```

```
      Currying.super.<init>();
      ()
    };
    def f1(a: Int, b: Int): Int = a.+(b);
    def f2(a: Int, b: Int): Int = a.+(b)
  }
}
```

As that output shows, I wrote the two functions `f1` and `f2` differently, but after the compiler's "uncurry" phase they end up looking the same.

> Things might look more interesting in the output if I had created a partially-applied function, but I'll leave that as an exercise for the reader.

Compiler phases

If you want to dig into this more, it can also help to know what the Scala compiler phases are. This command:

```
$ scalac -Xshow-phases
```

shows that the phases in Scala 2.11 are:

```
    phase name  id  description
    ----------  --  -----------
        parser   1  parse source into ASTs, perform simple desugaring
         namer   2  resolve names, attach symbols to named trees
packageobjects   3  load package objects
         typer   4  the meat and potatoes: type the trees
        patmat   5  translate match expressions
superaccessors   6  add super accessors in traits and nested classes
    extmethods   7  add extension methods for inline classes
       pickler   8  serialize symbol tables
```

```
     refchecks   9  reference/override checking, translate nested objects
       uncurry  10  uncurry, translate function values to anonymous classes
     tailcalls  11  replace tail calls by jumps
    specialize  12  @specialized-driven class and method specialization
 explicitouter  13  this refs to outer pointers
      erasure  14  erase types, add interfaces for traits
  posterasure  15  clean up erased inline classes
      lazyvals  16  allocate bitmaps, translate lazy vals into lazified defs
    lambdalift  17  move nested functions to top level
  constructors  18  move field definitions into constructors
       flatten  19  eliminate inner classes
         mixin  20  mixin composition
       cleanup  21  platform-specific cleanups, generate reflective calls
    delambdafy  22  remove lambdas
         icode  23  generate portable intermediate code
           jvm  24  generate JVM bytecode
      terminal  25  the last phase during a compilation run
```

As that shows, the "uncurry" phase "translates function values to anonymous classes."

Currying vs partially-applied functions

The concepts of currying and partially-applied functions are closely related, but they aren't exactly the same. As I wrote at the beginning, currying is defined like this:

> A function that takes multiple arguments can be translated into a series of function calls that each take a single argument.

This is particularly important in a language like Haskell, where all functions are technically curried functions. In Scala this is generally a theoretical thing that's good to know about, and it's good to know that you can create a curried function from an uncurried function, but these aren't "core" features you absolutely need to know to write code in Scala.

A partially-applied function on the other hand is a function that you manually create by supplying fewer parameters than the initial function defines. As I showed in this lesson, you can create the PAF `plus2` like this:

```
def plus(a: Int)(b: Int) = a + b
def plus2 = plus(2)(_)
```

and you can create `wrapWithDiv` as a PAF like this:

```
val wrapWithDiv = wrap("<div>")(_: String)("</div>")
```

If for some reason you want to partially-apply one parameter out of three, you can also do this:

```
def add(a: Int)(b: Int)(c: Int) = a + b + c
val add2NumbersTo10 = add(10)(_: Int)(_: Int)
```

So both concepts are related to multiple parameter groups, but in general, I use PAFs more often than I concern myself with curried functions.

Don't get bogged down in terminology

As I mentioned at the beginning of this lesson, don't get bogged down in the precise meaning of things like "curried functions." It is good to know how multiple input parameter groups work because it's a technique that is used a lot in Scala/FP, but don't get lost in worrying about the exact meaning of currying like I did. Understanding how multiple parameter groups work is the important thing.

Summary

This lesson covered the following topics:

- It provides a definition of *currying*
- It shows how to create partially-applied functions from functions that have (a) multiple parameter groups or (b) single parameter groups

It also shows how to create “curried” functions from regular functions, and provided a little look at how Scala gets these features to work with the JVM.

What’s next

I’ve covered a lot of Scala/FP background material so far, but occasionally I had to mix in a few `var` fields in my examples because that’s the only way to solve certain problems with the tools I’ve shown so far.

Well, no more of that.

In the next few lessons things are going to be fun, as I get to cover *recursion*. Once you understand recursive calls, I think you’ll find that they’re a natural way to think about writing iterative algorithms.

Once I cover recursion you’ll then be very close to handling many more FP concepts, the first of which will be how to handle “state” in FP applications. But to handle state in an FP manner, you’ll need to know how to write recursive functions …

See Also

Here are a few more resources related to currying and partially-applied functions.

- Daniel Westheide’s article, Currying and Partially Applied Functions is a good resource.

These discussions on StackOverflow and StackExchange also provide a little more insight:

- With curried functions you get easier reuse of more abstract functions, since you get to specialize.
- “It’s common to mistake partial function application for currying … I’ve almost never seen anyone use currying in practice. Partial function application on the other hand is quite useful in many languages.”
- “There is a slight difference between currying and partial application,

although they’re closely related; since they’re often mixed together, I’ll deal with both terms.”

29

Recursion: Introduction

As you may have noticed from this book's index, you're about to jump into a series of lessons on recursive programming. I separated this text into a series of small lessons to make the content easier to read initially, and then easier to refer to later.

Please note that some of these lessons may be overkill for some people. This is, after all, the first draft of this book, and I'm trying to find the best ways to teach recursive programming. I start by reviewing the `List` class, then show a straightforward, "Here's how to write a recursive function" lesson. After that I add a few more lessons to explain recursion in different ways.

If at any point you feel like you understand how to write recursive functions, feel free to skip any or all of these lessons. You can always come back to them later if you need to.

30

Recursion: Motivation

> *"To iterate is human, to recurse divine."*
>
> L. Peter Deutsch

What is recursion?

Before getting into the motivation to use recursion, a great question is, "What is recursion?"

Simply stated, a *recursive function* is a function that calls itself. That's it.

As you'll see in this lesson, a common use of recursive functions is to iterate over the elements in a list.

Why do I need to write recursive functions?

The next question that usually comes up right about now is, "Why do I need to write recursive functions? Why can't I use `for` loops to iterate over lists?"

The short answer is that algorithms that use `for` loops require the use of `var` fields, and as you know from our rules, functional programmers don't use `var` fields.

(Read on for the longer answer.)

If you had `var` fields

Of course if you *could* use mutable variables in your programming language, you might write a "sum the integers in a list" algorithm like this:

```
def sum(xs: List[Int]): Int = {
    var sum = 0
    for (x <- xs) {
        sum += x
    }
    sum
}
```

That algorithm uses a `var` field named `sum` and a `for` loop to iterate through every element in the given list to calculate the sum of those integers. From an imperative programming standpoint, there's nothing wrong with this code. I wrote imperative code like this in Java for more than fifteen years.

But from a functional programmer's point of view, there are several problems with this code.

Problem 1: We can only keep so much in our brains

One problem is that reading a lot of custom `for` loops dulls your brain.

As an OOP/imperative programmer I never noticed it, but if you *think about the way you thought* when you read that function, one of the first things you thought is, "Hmm, here's a `var` field named `sum`, so Al is probably going to modify that field in the rest of the algorithm." Then you thought, "Okay, here's a `for` loop … he's looping over `xs` … ah, yes, he's using `+=`, so this really is a 'sum' loop, so that variable name makes sense." Once you learn FP — or even if you just learn the methods available on Scala collections classes — you realize *that's a lot of thinking* about a little custom `for` loop.

If you're like me a few years ago, you may be thinking that what I just wrote is overkill. You probably look at mutable variables and `for` loops all the time. But studies show that we can only keep *just so much* information in our brains at one time, therefore:

- The less information we *have* to keep in there is a win, and
- Boilerplate `for` loop code is a waste of our brain's RAM

Maybe this seems like a small, subtle win at the moment, but speaking from my own experience, anything I can do to keep my brain's RAM free for important things is a win.

See the Wikipedia article, The Magical Number 7 (Plus or Minus 2) for a good discussion on how much information we humans can keep in our brains at any one time.

Problem #2: It's not algebraic

Another problem is that this code doesn't look or feel like algebra. I discussed this in the "Functional Programming is Like Algebra" lesson, so I won't repeat that discussion here.

Problem #3: There are no `var` fields in FP

Of course from our perspective as functional programmers, the *huge* problem with this code is that it requires a `var` field, and Scala/FP developers don't use those. A `var` field is a crutch, and the best thing you can do to expedite your FP education is to completely forget that they exist.

In my own FP experience, I learned that there's a different way to solve iterative problems once I let go of `var` fields and `for` loops.

What to do?

Because we can't use `var` fields, we need to look at a different tool to solve problems like this. That tool is *recursion*.

If you're like me, at first you'll *need* to write recursive functions (because that's all you can do), but after a while you'll *want* to write recursive functions.

31

Recursion: Let’s Look at Lists

> *“In computer science, a linked list is a linear collection of data elements, called nodes, each pointing to the next node by means of a pointer.”*
>
> Wikipedia’s Linked List entry

Visualizing lists

Because the `List` data structure — and the *head* and *tail* components of a `List` — are so important to recursion, it helps to visualize what a list and its head and tail components look like. Figure 31.1 shows one way to visualize a `List`.

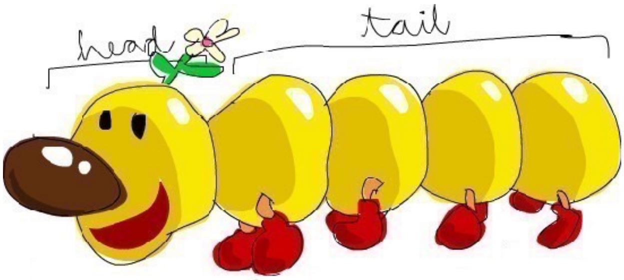


Figure 31.1: One way to visualize the head and tail elements of a list.

This creative imagery comes from the online version of “Learn You a Haskell for Great Good”, and it does a great job of imprinting the concept of head and tail components of a list into your brain. As shown, the “head” component is simply the first element in the list, and the “tail” is the rest of the list.

A slightly more technical way to visualize the head and tail of a list is shown in Figure 31.2.

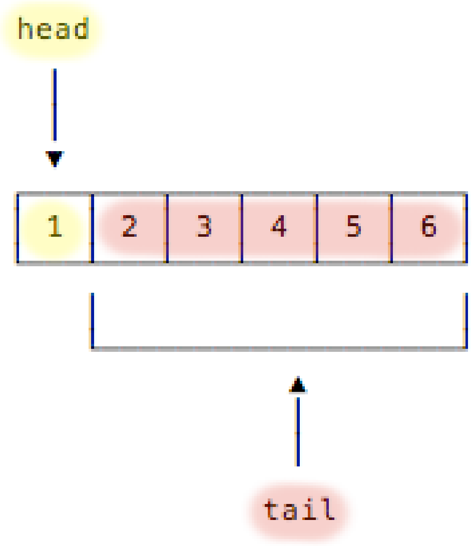


Figure 31.2: A slightly more technical way to visualize a list.

An even more accurate way to show this is with a `Nil` value at the end of the `List`, as shown in Figure 31.3, because that's what it really looks like:

Linked lists and "cons" cells

To be clear, the `List` that I'm talking about is a *linked list* — scala.collection.immutable.List, which is the default list you get if you type `List` in your IDE or the REPL. This `List` is a series of cells, where each cell contains two things: (a) a value, and (b) a pointer to the next cell. This is shown in Figure 31.4.

As shown, the last cell in a linked list contains the `Nil` value. The `Nil` in the last cell is *very* important: it's how your recursive Scala code will know when it has reached the end of a `List`.

When drawing a list like this, Figure 31.5 clearly shows the head element of a list, and Figure 31.6 shows the tail elements.

Just like Haskell — and Lisp before it — the default Scala `List` works with these head and tail components, and I'll use them extensively in the examples that follow.

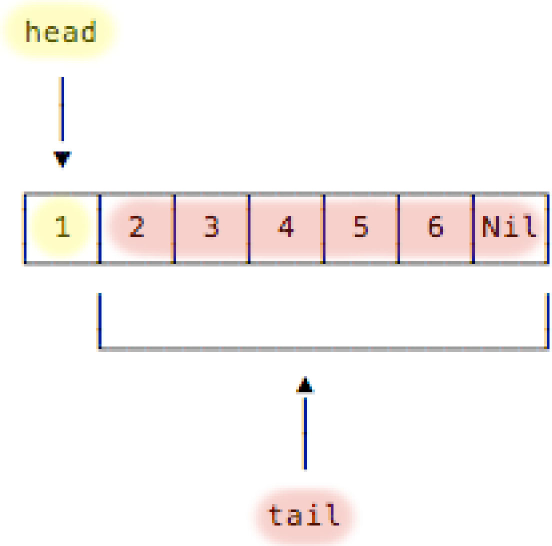


Figure 31.3: A more accurate way to visualize a list.

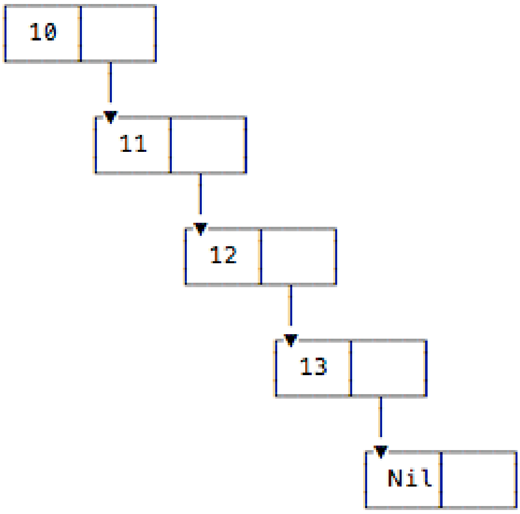


Figure 31.4: An accurate depiction of a linked list.

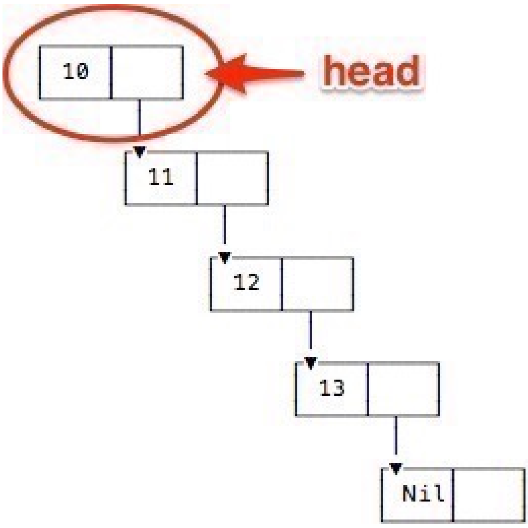


Figure 31.5: The head element of a list.

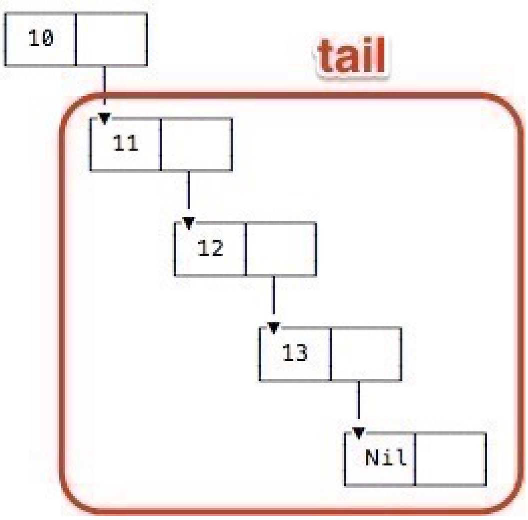


Figure 31.6: The tail elements of a list.

> For historical reasons these cells are known as "cons cells." That name comes from Lisp, and if you like history, you can read more about it on Wikipedia.

Note 1: The empty `List`

As a first note about `List`s, a `List` with no elements in it is an *empty list*. An empty `List` contains only one cell, and that cell contains a `Nil` element, as shown in Figure 31.7.

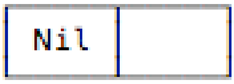


Figure 31.7: A list with no elements contains only one cell, which contains a `Nil` *element.*

You can create an empty `List` in Scala in two ways:

```
scala> val empty = List()
empty: List[Nothing] = List()

scala> val empty = Nil
empty: scala.collection.immutable.Nil.type = List()
```

Because I haven't given those lists a data type (like `Int`), the results look a little different, but if I add a type to those expressions, you'll see that the result is exactly the same:

```
scala> val empty1: List[Int] = List()
empty: List[Int] = List()

scala> val empty2: List[Int] = Nil
empty: List[Int] = List()

scala> empty1 == empty2
res0: Boolean = true
```

In summary:

```
List() == Nil
```

Note 2: Several ways to create `Lists`

There are several ways to create non-empty `Lists` in Scala, but for the most part I'll use two approaches. First, here's a technique you're probably already familiar with:

```
val list = List(1,2,3)
```

Second, this is an approach you may not have seen yet:

```
val list = 1 :: 2 :: 3 :: Nil
```

These two techniques result in the exact same `List[Int]`, which you can see in the REPL:

```
scala> val list1 = List(1,2,3)
list: List[Int] = List(1, 2, 3)

scala> val list2 = 1 :: 2 :: 3 :: Nil
list: List[Int] = List(1, 2, 3)

scala> list1 == list2
res1: Boolean = true
```

The second approach is known as using "cons cells." As you can see, it's a very literal approach to creating a `List`, where you specify each element in the `List`, including the `Nil` element, which must be in the last position. If you forget the `Nil` element at the end, the Scala compiler will bark at you:

```
scala> val list = 1 :: 2 :: 3
<console>:10: error: value :: is not a member of Int
       val list = 1 :: 2 :: 3
                         ^
```

I show this because it's important — very important — to know that the last element in a `List` *must* be the `Nil` element. (I like to say that the `Nil` element is to a `List` as a caboose is to a train.) We're going to take advantage of this knowledge as we write our first recursive function.

32

Recursion: How to Write a 'sum' Function

With all of the images of the previous lesson firmly ingrained in your brain, let's write a `sum` function using recursion!

Source code

You can follow along with the source code in this lesson by cloning my project from this Github URL:

- My Recursive Sum example

Sketching the `sum` function signature

Given a `List` of integers, such as this one:

```
val list = List(1, 2, 3, 4)
```

let's start tackling the problem in the usual way, by thinking, "Write the function signature first."

What do we know about the `sum` function we want to write? Well, we know a couple of things:

- It will take a list of integers as input
- Because it returns a sum of those integers, the function will return a single value, an `Int`

Armed with only those two pieces of information, I can sketch the signature for a `sum` function like this:

```
def sum(list: List[Int]): Int = ???
```

> Note: For the purposes of this exercise I'm assuming that the integer values will be small, and the list size will also be small. That way we don't have to worry about all of the `Int`s adding up to a `Long`.

The `sum` function body

At this point a functional programmer will think of a "sum" algorithm as follows:

1. If the `sum` function is given an empty list of integers, it should return `0`. (Because the sum of nothing is zero.)
2. Otherwise, if the list is *not* empty, the result of the function is the combination of (a) the value of its head element (`1`, in this case), and (b) the sum of the remaining elements in the list (`2,3,4`).

A slight restatement of that second sentence is:

> "The sum of a list of integers is the sum of the *head* element, plus the sum of the *tail* elements."

As Eckhart Tolle is fond of saying, "That statement is true, is it not?"

(Yes, it is.)

Thinking about a `List` in terms of its head and tail elements is a standard way of thinking when writing recursive functions.

Now that we have a little idea of how to *think* about the problem recursively, let's see how to implement those sentences in Scala code.

Implementing the first sentence in code

The first sentence above states:

> If the `sum` function is given an empty list of integers, it should return `0`.

Recursive Scala functions are often implemented using `match` expressions. Using (a) that information and (b) remembering that an empty list contains only the `Nil` element, you can start writing the body of the `sum` function like this:

```
def sum(list: List[Int]): Int = list match {
    case Nil => 0
```

This is a Scala way of saying, "If the `List` is empty, return `0`." If you're comfortable with `match` expressions and the `List` class, I think you'll agree that this makes sense.

Note 1: Using `return`

If you prefer using `return` statements at this point in your programming career, you can write that code like this:

```
def sum(list: List[Int]): Int = list match {
    case Nil => return 0
```

Because a pure function doesn't "return" a value as much as it "evaluates" to a result, you'll want to quickly drop `return` from your vocabulary, but … I also understand that using `return` can help when you first start writing recursive functions.

Note 2: Using if/then instead

You can also write this function using an if/then expression, but because *pattern matching* is such a big part of functional programming, I prefer `match` expressions.

Note 3: Can also use `List()`

Because `Nil` is equivalent to `List()`, you can also write that `case` expression like this:

```
case List() => 0
```

However, most functional programmers use `Nil`, and I'll continue to use `Nil` in this lesson.

Implementing the second sentence in code

That `case` expression is a Scala/FP implementation of the first sentence, so let's move on to the second sentence.

The second sentence says, "If the list is *not* empty, the result of the algorithm is the combination of (a) the value of its head element, and (b) the sum of its tail elements."

To split the list into head and tail components, I start writing the second `case` expression like this:

```
case head :: tail => ???
```

If you know your `case` expressions, you know that if `sum` is given a list like `List(1,2,3,4)`, this pattern has the result of assigning `head` to the value `1`, and assigning `tail` the value `List(2,3,4)`:

```
head = 1
tail = List(2,3,4)
```

> (If you don't know your `case` expressions, please refer to the match/case lessons in Chapter 3 of the Scala Cookbook.)

This `case` expression is a start, but how do we finish it? Again I go back to the second sentence:

> If the list is *not* empty, the result of the algorithm is the combination of (a) the value of its head element, and (b) the sum of the tail elements.

The "value of its head element" is easy to add to the `case` expression:

```
case head :: tail => head ...
```

But then what? As the sentence says, "the value of its head element, and the sum of the tail elements," which tells us we'll be adding *something* to `head`:

```
case head :: tail => head + ???
```

What are we adding to `head`? *The sum of the list's tail elements.* Hmm, now how can we get the sum of a list of tail elements? How about this:

```
case head :: tail => head + sum(tail)
```

Whoa. That code is a straightforward implementation of the sentence, isn't it?

(I'll pause here to let that sink in.)

If you combine this new `case` expression with the existing code, you get the following `sum` function:

```
def sum(list: List[Int]): Int = list match {
    case Nil => 0
    case head :: tail => head + sum(tail)
}
```

And that is a recursive "sum the integers in a `List`" function in Scala/FP. No `var`'s, no `for` loop.

A note on those names

If you're new to `case` expressions, it's important to note that the `head` and `tail` variable names in the second `case` expression can be anything you want. I wrote it like this:

```
case head :: tail => head + sum(tail)
```

but I could have written this:

```
case h :: t => h + sum(t)
```

or this:

```
case x :: xs => x + sum(xs)
```

This last example uses variable names that are commonly used with FP, lists, and recursive programming. When working with a list, a single element is often referred to as `x`, and multiple elements are referred to as `xs`. It's a way of indicating that `x` is singular and `xs` is plural, like referring to a single "pizza" or multiple "pizzas." With lists, the head element is definitely singular, while the tail can contain one or more elements. I'll generally use this naming convention in this book.

Proof that `sum` works

To demonstrate that `sum` works, you can clone my RecursiveSum project on Github — which uses ScalaTest to test `sum` — or you can copy the following source code that extends a Scala `App` to test `sum`:

```
object RecursiveSum extends App {

    def sum(list: List[Int]): Int = list match {
        case Nil => 0
        case x :: xs => x + sum(xs)
    }

    val list = List(1, 2, 3, 4)
    val sum = sum(list)
    println(sum)
```

```
}
```

When you run this application you should see the output, `10`. If so, congratulations on your first recursive function!

“That’s great,” you say, “but how exactly did that end up printing `10`?”

To which I say, “Excellent question. Let’s dig into that!”

> As I’ve noted before, I tend to write verbose code that’s hopefully easy to understand, especially in books, but you can shrink the last three lines of code to this, if you prefer:
>
> ```
> println(sum(List(1,2,3,4)))
> ```

33

Recursion: How Recursive Function Calls Work

An important point to understand about recursive function calls is that just as they "wind up" as they are called repeatedly, they "unwind" rapidly when the function's end condition is reached.

In the case of the `sum` function, the end condition is reached when the `Nil` element in a `List` is reached. When `sum` gets to the `Nil` element, this pattern of the `match` expression is matched:

```
case Nil => 0
```

Because this line simply returns `0`, there are no more recursive calls to `sum`. This is a typical way of ending the recursion when operating on all elements of a `List` in recursive algorithms.

Lists end with `Nil`

As I wrote in the earlier `List` lesson, a literal way to create a `List` is like this:

```
1 :: 2 :: 3 :: 4 :: Nil
```

This is a reminder that with any Scala `List` you are guaranteed that the last `List` element is `Nil`. Therefore, if your algorithm is going to operate on the entire list, you should use:

```
case Nil => ???
```

as your function's end condition.

This is the first clue about how the unfolding process works.

> Note 1: This is a feature of the Scala `List` class. You'll have to change the approach if you work with other sequential collection classes like `Vector`, `ArrayBuffer`, etc. (More on this later in the book.)

> Note 2: Examples of functions that work on every element in a list are `map`, `filter`, `foreach`, `sum`, `product`, and many more. Examples of functions that *don't* operate on every list element are `take` and `takeWhile`.

Understanding how the `sum` example ran

A good way to understand how the `sum` function example ran is to add `println` statements inside the `case` expressions.

First, change the `sum` function to look like this:

```
def sum(list: List[Int]): Int = list match {
    case Nil => {
        println("case1: Nil was matched")
        0
    }
    case head :: tail => {
        println(s"case2: head = $head, tail = $tail")
        head + sum(tail)
    }
}
```

Now when you run it again with a `List(1,2,3,4)` as its input parameter, you'll see this output:

```
case2: head = 1, tail = List(2, 3, 4)
case2: head = 2, tail = List(3, 4)
case2: head = 3, tail = List(4)
case2: head = 4, tail = List()
case1: Nil was matched
```

That output shows that `sum` is called repeatedly until the list is reduced to `List()` (which is the same as `Nil`). When `List()` is passed to `sum`, the first `case` is matched and the recursive calls to `sum` come to an end. (I'll demonstrate this visually in the next lesson.)

The book, Land of Lisp states, "recursive functions are list eaters," and this output shows why that statement is true.

How the recursion works ("going down")

Keeping in mind that `List(1,2,3,4)` is the same as `1::2::3::4::Nil`, you can read the output like this:

1. The first time `sum` is called, the `match` expression sees that the given `List` doesn't match the `Nil` element, so control flows to the second `case` statement.
2. The second `case` statement matches the `List` pattern, then splits the incoming list of `1::2::3::4::Nil` into (a) a head element of `1` and

    (b) the remainder of the list, `2::3::4::Nil`. The remainder — the tail — is then passed into another `sum` function call.

3. A new instance of `sum` receives the list `2::3::4::Nil`. It sees that this list does not match the `Nil` element, so control flows to the second `case` statement.
4. That statement matches the `List` pattern, then splits the list into a head element of `2` and a tail of `3::4::Nil`. The tail is passed as an input parameter to another `sum` call.
5. A new instance of `sum` receives the list `3::4::Nil`. This list does not match the `Nil` element, so control passes to the second `case` statement.
6. The list matches the pattern of the second `case` statement, which splits the list into a head element of `3` and a tail of `4::Nil`. The tail is passed as an input parameter to another `sum` call.
7. A new instance of `sum` receives the list `4::Nil`, sees that it does not match `Nil`, and passes control to the second `case` statement.

8. The list matches the pattern of the second `case` statement. The list is split into a head element of `4` a tail of `Nil`. The tail is passed to another `sum` function call.
9. The new instance of `sum` receives `Nil` as an input parameter, and sees that it *does* match the `Nil` pattern in the first `case` expression. At this point the first `case` expression is evaluated.
10. The first `case` expression returns the value `0`. This marks the end of the recursive calls.

At this point — when the first `case` expression of this `sum` instance returns `0` — all of the recursive calls "unwind" until the very first `sum` instance returns its answer to the code that called it.

How the unwinding works ("coming back up")

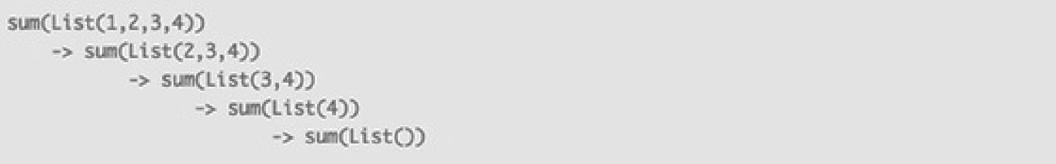
That description gives you an idea of how the recursive `sum` function calls work until they reach the end condition. Here's a description of what happens *after* the end condition is reached:

1. The last `sum` instance — the one that received `List()` — returns `0`. This happens because `List()` matches `Nil` in the first `case` expression.
2. This returns control to the previous `sum` instance. The second `case` expression of that `sum` function has `return 4 + sum(Nil)` as its return value. This is reduced to `return 4 + 0`, so this instance returns `4`. (I didn't use a `return` statement in the code, but it's easier to read this now if I say "return.")
3. Again, this returns control to the previous `sum` instance. That `sum` instance has `return 3 + sum(List(4))` as the result of its second `case` expression. You just saw that `sum(List(4))` returns `4`, so this `case` expression evaluates to `return 3 + 4`, or `7`.
4. Control is returned to the previous `sum` instance. Its second `case` expression has `return 2 + sum(List(3,4))` as its result. You just saw that `sum(List(3,4))` returns `7`, so this expression evaluates to `return 2 + 7`, or `9`.
5. Finally, control is returned to the original `sum` function call. Its second `case` expression is `return 1 + sum(List(2,3,4))`. You just saw that

`sum(List(2,3,4))` returns `9`, so this call is reduced to `return 1 + 9`, or `10`. This value is returned to whatever code called the first `sum` instance.

Initial visuals of how the recursion works

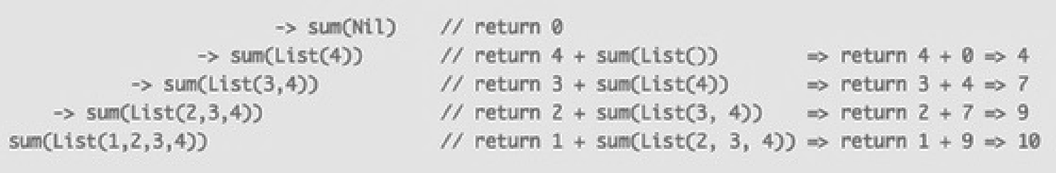
One way to visualize how the recursive `sum` function calls work — the "going down" part — is shown in Figure 33.1.



```
sum(List(1,2,3,4))
    -> sum(List(2,3,4))
           -> sum(List(3,4))
                 -> sum(List(4))
                        -> sum(List())
```

Figure 33.1: How the original `sum` *call leads to another, then to another …*

After that, when the end condition is reached, the "coming back up" part — what I call the unwinding process — is shown in Figure 33.2.



```
                        -> sum(Nil)    // return 0
                 -> sum(List(4))       // return 4 + sum(List())        => return 4 + 0 => 4
           -> sum(List(3,4))           // return 3 + sum(List(4))       => return 3 + 4 => 7
    -> sum(List(2,3,4))                // return 2 + sum(List(3, 4))    => return 2 + 7 => 9
sum(List(1,2,3,4))                     // return 1 + sum(List(2, 3, 4)) => return 1 + 9 => 10
```

Figure 33.2: How `sum` *function calls unwind, starting with the last* `sum` *call.*

If this isn't clear, fear not, in the next lesson I'll show a few more visual examples of how this works.

34

Visualizing the Recursive sum Function

Another way to view recursion is with visual diagrams. To demonstrate this, I'll use the rectangular symbol shown in Figure 34.1 to represent a function.

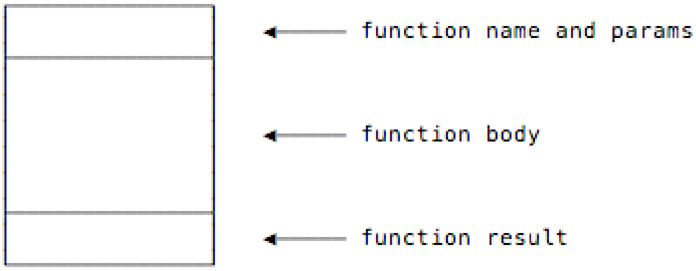


Figure 34.1: This rectangular symbol will be used to represent functions in this lesson.

The first step

Using that symbol and a list with only three elements, Figure 34.2 shows a representation of the first `sum` function call.

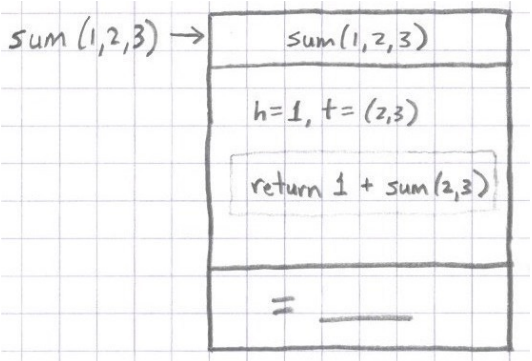


Figure 34.2: A visual representation of the first `sum` *call.*

The top cell in the rectangle indicates that this first instance of `sum` is called with the parameters `1,2,3`. Note that I'm leaving the "`List`" name off of these diagrams to make them more readable.

The body of the function is shown in the middle region of the symbol, and it's shown as `return 1 + sum(2,3)`. As I mentioned before, you don't normally use the `return` keyword with Scala/FP functions, but in this case it makes the diagram more clear.

In the bottom region of the symbol I've left room for the final return value of the function. At this time we don't know what the function will return, so for now I just leave that spot empty.

The next steps

For the next step of the diagram, assume that the first `sum` function call receives the parameter list `(1,2,3)`, and its body now calls a new instance of `sum` with the input parameter `sum(2,3)` (or `sum(List(2,3))`, if you prefer). You can imagine the second `case` expression separating the `List` into head and tail elements, as shown in Figure 34.3.

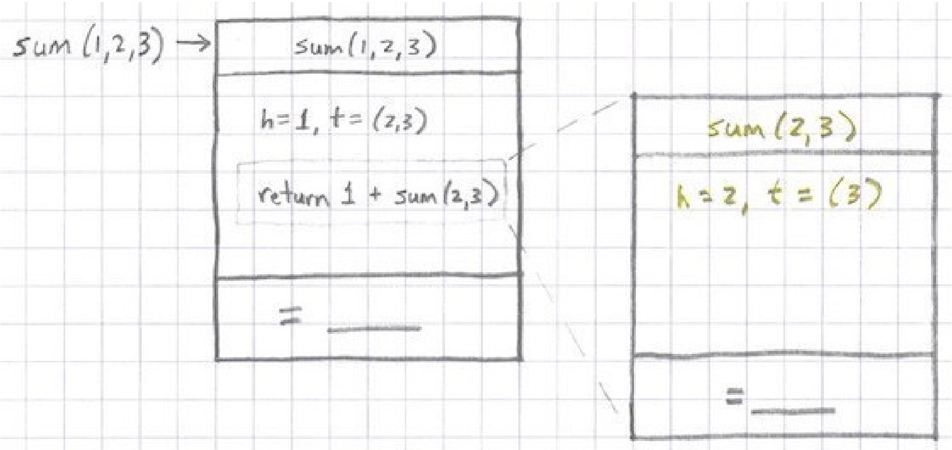


Figure 34.3: The first `sum` *function invokes a second* `sum` *function call.*

Then this `sum` instance makes a recursive call to another `sum` instance, as shown in Figure 34.4.

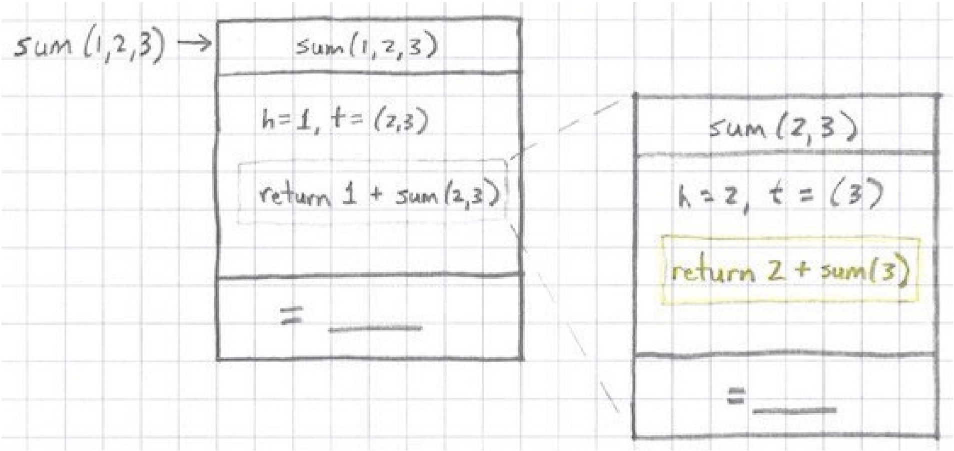


Figure 34.4: The second `sum` *function call begins to invoke the third* `sum` *instance.*

Again I leave the return value of this function empty because I don't know what it will be until its `sum` call returns.

It's important to be clear that these two function calls are completely different instances of `sum`. They have their own input parameter lists, local variables, and return values. It's just as if you had two different functions, one named `sum3elements` and one named `sum2elements`, as shown in Figure 34.5.

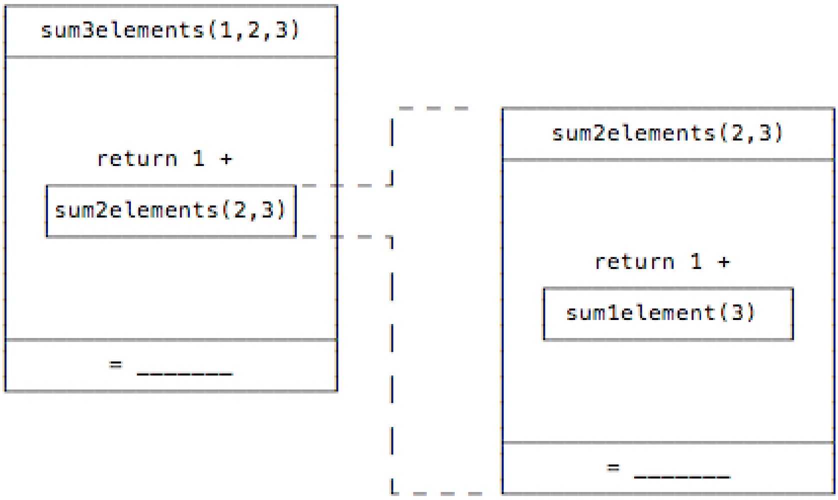


Figure 34.5: One `sum` *function calling another* `sum` *instance is just like calling a different function.*

Just as the variables inside of `sum3elements` and `sum2elements` have completely different scope, the variables in two different instances of `sum` also have completely different scope.

Getting back to the `sum` example, you can now imagine that the next step will proceed just like the previous one, as shown in Figure 34.6.

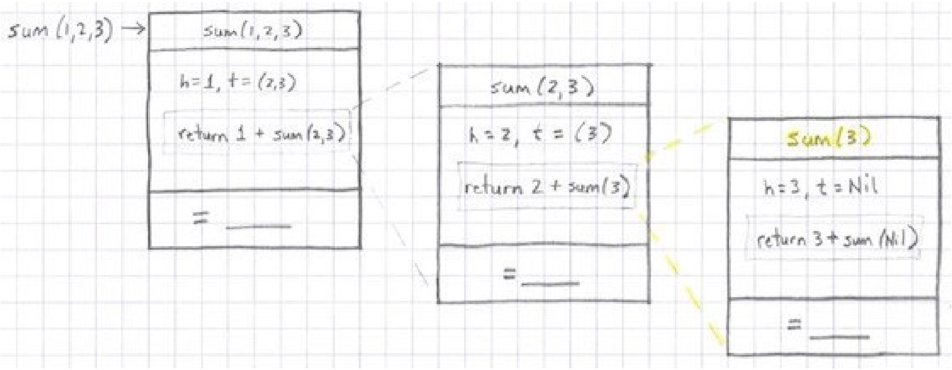


Figure 34.6: The third `sum` *function has now been called.*

The last recursive `sum` *call*

Now we're at the point where we make the last recursive call to `sum`. In this case, because `3` was the last integer in the list, a new instance of `sum` is called with the `Nil` value. This is shown in Figure 34.7.

With this last `sum` call, the `Nil` input parameter matches the first `case` expression, and that expression simply returns `0`. So now we can fill in the return value for this function, as shown in Figure 34.8.

Now this `sum` instance returns `0` back to the previous `sum` instance, as shown in Figure 34.9.

The result of this function call is `3 + 0` (which is `3`), so you can fill in its return value, and then flow it back to the previous `sum` call. This is shown in Figure 34.10.

The result of this function call is `2 + 3` (`5`), so that result can flow back to the previous function call, as shown in Figure 34.11.

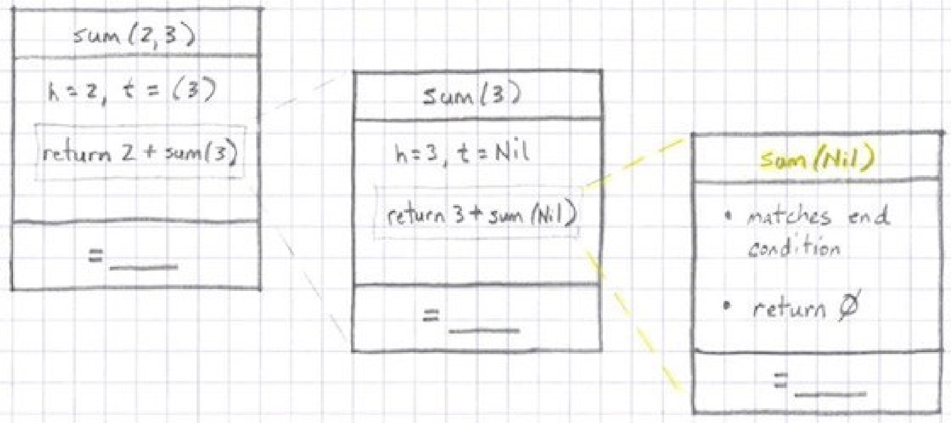


Figure 34.7: `Nil` *is passed into the final* `sum` *function call.*

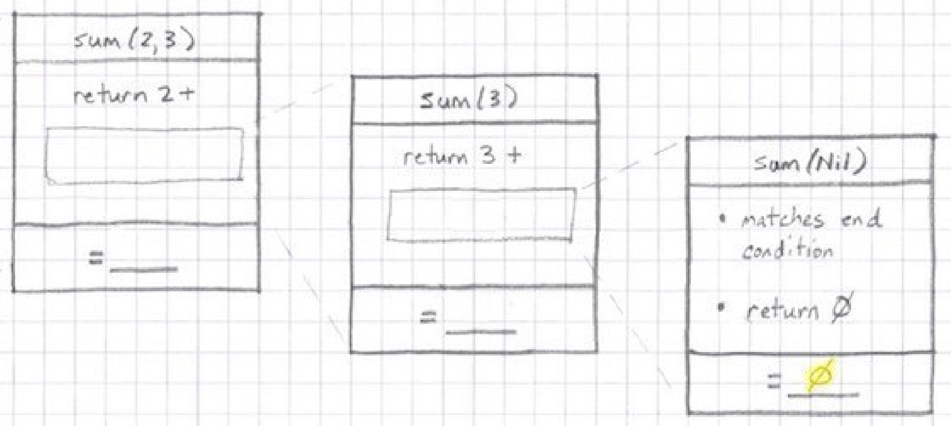


Figure 34.8: The return value of the last `sum` *call is* `0`*.*

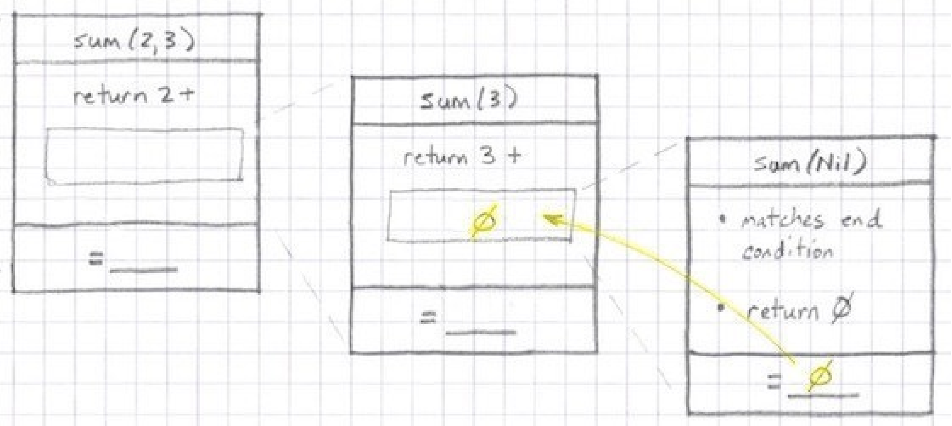


Figure 34.9: `0` *is returned back to the previous* `sum` *call.*

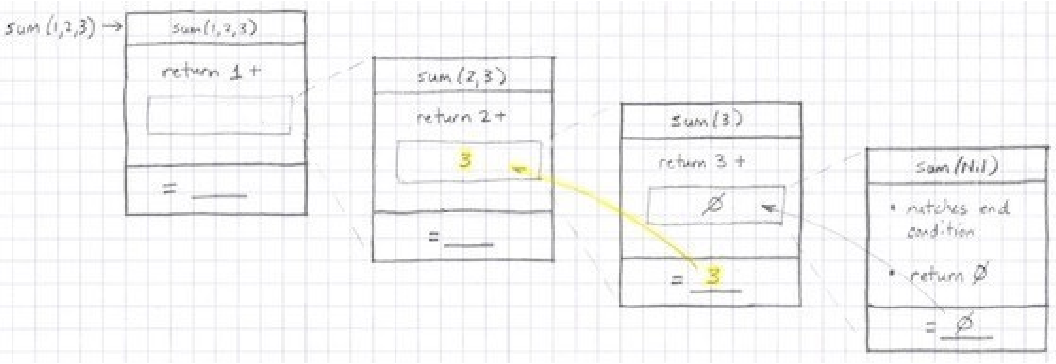


Figure 34.10: The third `sum` call returns to the second.

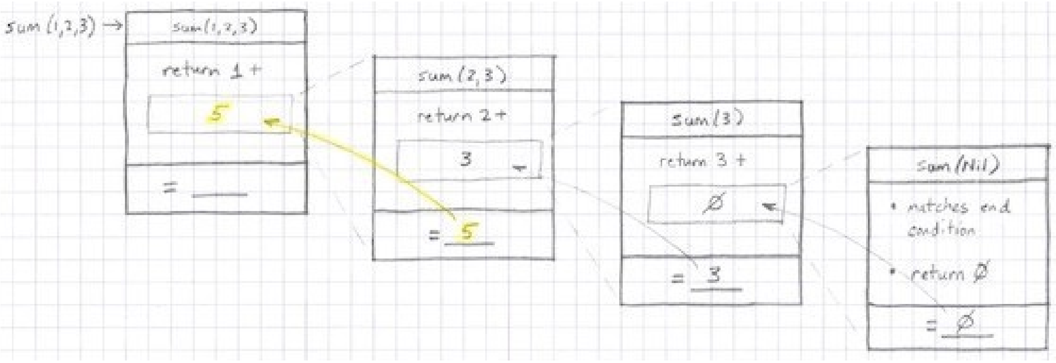


Figure 34.11: The second `sum` call returns to the first.

Finally, the result of this `sum` instance is `1 + 5` (`6`). This was the first `sum` function call, so it returns the value `6` back to whoever called it, as shown in Figure 34.12.

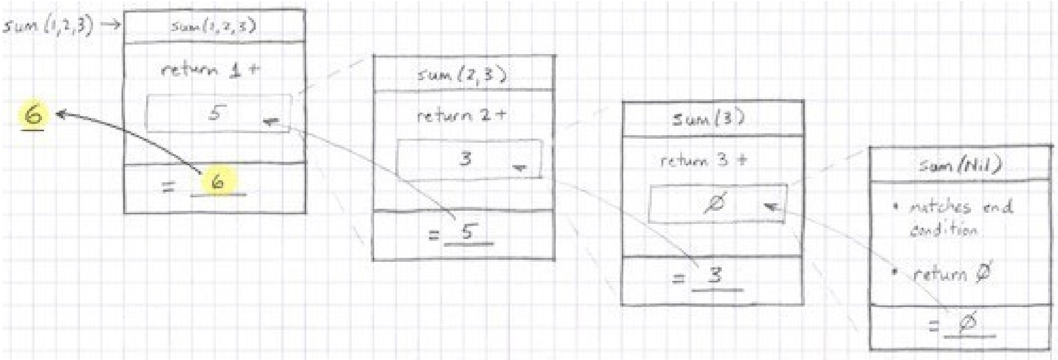


Figure 34.12: The first `sum` *call returns to the final result.*

Other visualizations

There are other ways to draw recursive function calls. Another nice approach is to use a modified version of a UML "Sequence Diagram," as shown in Figure 34.13. Note that in this diagram "time" flows from the top to the bottom.

This diagram shows that the `main` method calls `sum` with the parameter `List(1,2,3)`, where I again leave off the `List` part; it calls `sum(2,3)`, and so on, until the `Nil` case is reached, at which point the return values flow back from right to left, eventually returning `6` back to the `main` method.

You can write the return values like that, or with some form of the function's equation, as shown in Figure 34.14.

Personally, I use whatever diagram seems to help the most.

Summary

Those are some visual examples of how recursive function calls work. If you find yourself struggling to understand how recursion works, I hope these diagrams are helpful.

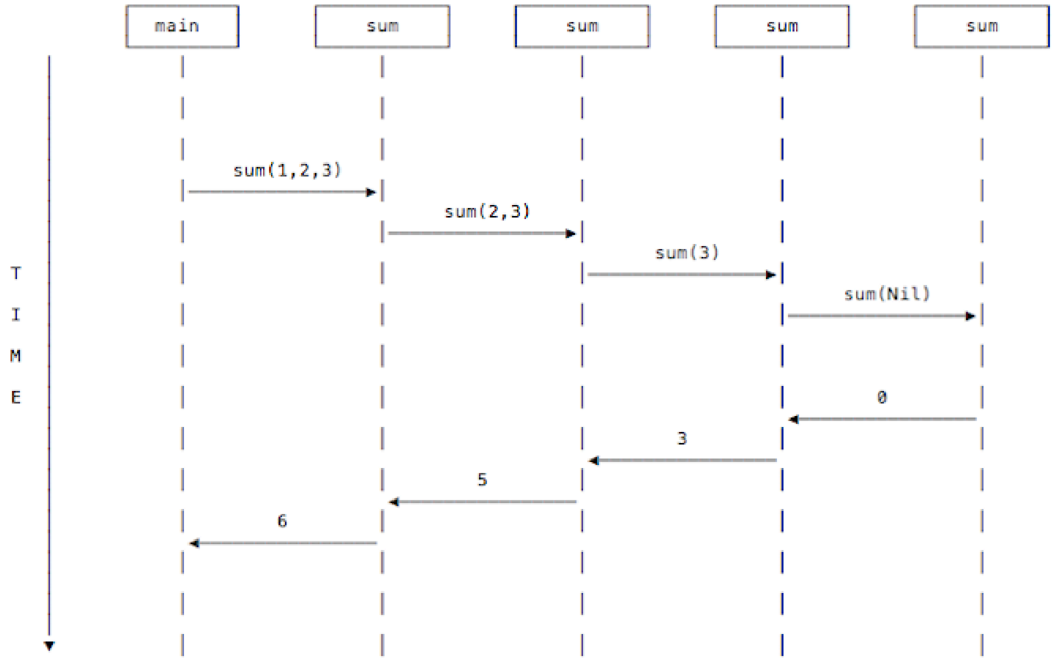


Figure 34.13: The `sum` *function calls can be shown using a UML Sequence Diagram.*

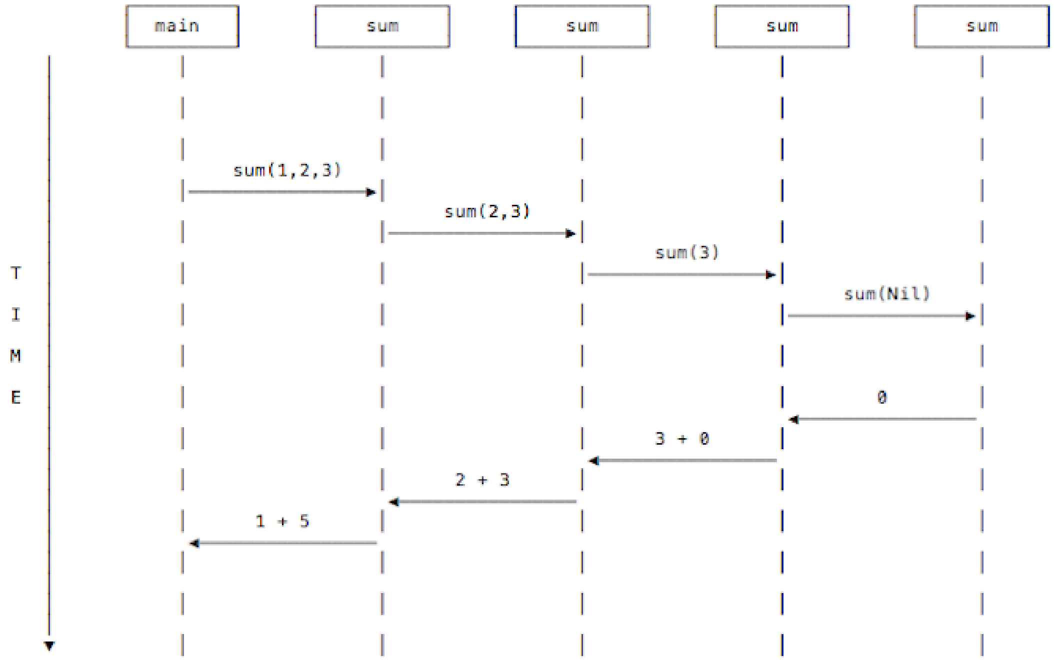


Figure 34.14: Writing the function return values as equations.

35

Recursion: A Conversation Between Two Developers

As an homage to one of my favorite Lisp books — an early version of what is now The Little Schemer — this lesson shows a little question and answer interaction that you can imagine happening between two Scala programmers.

Given this `sum` function:

```
def sum(list: List[Int]): Int = list match {
    case Nil => 0
    case x :: xs => x + sum(xs)
}
```

I hope this "conversation" will help drive home some of the points about how recursion works:

Person 1	Person 2
What is this? `val x = List(1,2,3,4)`	An expression that defines a `List[Int]`, which in this case contains the integers `1` through `4`. The expression binds that list to the variable `x`.
And what is this? `x.head`	The first element of the list `x`, which is `1`.
How about this? `x.tail`	That's the remaining elements in the list `x`, which is `List(2,3,4)`.
How about this: `x.tail.head`	It is the number `2`.

Person 1	Person 2
How did you come up with that?	`x.tail` is `List(2,3,4)`, and `List(2,3,4).head` is the first element of that list, or `2`.
How about this: `x.tail.tail`	That’s `List(3,4)`.
Explain, please.	`x.tail` is `List(2,3,4)`, and then `List(2,3,4).tail` is `List(3,4)`.
Are you ready for more?	Yes, please.
Given the definition of our `sum` function, explain the first step in: `sum(List(1,2,3))`.	The `sum` function receives `List(1,2,3)`. This does not match the `Nil` case, but does match the second case, where `x` is assigned to `1` and `xs` is `List(2,3)`.
Then what happens?	A new instance of `sum` is called with the parameter `List(2,3)`.
And then?	A new instance of `sum` receives the input parameter `List(2,3)`. This does not match the `Nil` case, but does match the second case, where `x` is assigned to `2` and `xs` is `List(3)`.
Please continue.	`sum` is called with the parameter `List(3)`.
Go on.	A new instance of `sum` receives `List(3)`. This does not match the `Nil` case, but does match the second case, where `x` is assigned to `3` and `xs` is `List()`.
Don’t stop now.	`sum` is called with the parameter `List()`.
What happens inside this instance of `sum`?	It receives `List()`. This is the same as `Nil`, so it matches the first case.
Cool. Something different. Now what happens?	That case returns `0`.

Person 1	Person 2
Ah, finally a return value!	You’re telling me.
Okay, so now what happens?	This ends the recursion, and then the recursive calls unwind, as described in the previous lesson.

36

Recursion: Thinking Recursively

> *"To understand recursion, one must first understand recursion."*
>
> Stephen Hawking

Goal

This lesson has one primary goal: to show that the thought process followed in writing the `sum` function follows a common recursive programming "pattern." Indeed, when you write recursive functions you'll generally follow the three-step process shown in this lesson.

I don't want to make this too formulaic, but the reality is that if you follow these three steps in your thinking, it will make it easier to write recursive functions, especially when you first start.

The general recursive thought process (the "three steps")

As I mentioned in the previous lessons, when I sit down to write a recursive function, I think of three things:

- What is the function signature?
- What is the end condition for this algorithm?
- What is the actual algorithm? For example, if I'm processing all of the elements in a `List`, what does my algorithm do when the function receives a non-empty `List`?

Let's take a deep dive into each step in the process to make more sense of these descriptions.

Step 1: What is the function signature?

Once I know that I'm going to write a recursive function, the first thing I ask myself is, "What is the signature of this function?"

If you can describe the function verbally, you should find that you know (a) the parameters that will be passed into the function and (b) what the function will return. In fact, if you *don't* know these things, you're probably not ready to write the function yet.

The sum function

In the `sum` function the algorithm is to add all of the integers in a given list together to return a single integer result. Therefore, because I know the function takes a list of integers as its input, I can start sketching the function signature like this:

```
def sum(list: List[Int]) ...
```

Because the description also tells me that the function returns an `Int` result, I add the function's return type:

```
def sum(list: List[Int]): Int = ???
```

This is the Scala way to say that "the `sum` function takes a list of integers and returns an integer result," which is what I want. In FP, sketching the function signature is often half of the battle, so this is actually a big step.

Step 2: How will this algorithm end?

The next thing I usually think about is, "How will this algorithm end? What is its end condition?"

Because a recursive function like `sum` keeps calling itself over and over, it's of the utmost importance that there is an end case. If a recursive algorithm doesn't have an end condition, it will keep calling itself as fast as possible until either (a) your program crashes with a `StackOverflowError`, or (b) your computer's CPU gets extraordinarily hot. Therefore, I offer this tip:

> Always have an end condition, and write it as soon as possible.

In the `sum` algorithm you know that you have a `List`, and you want to march through the entire `List` to add up the values of all of its elements. You may not know it at this point in your recursive programming career, but right away this statement is a big hint about the end condition. Because (a) you know that you're working with a `List`, (b) you want to operate on the entire `List`, and (c) a `List` ends with the `Nil` element, (d) you can begin to write the end condition `case` expression like this:

```
case Nil => ???
```

To be clear, this end condition is correct because you're working with a `List`, and you know that the algorithm will operate on the entire `List`. Because the `Nil` element is to a `List` as a caboose is to a train, you're guaranteed that it's always the last element of the `List`.

> Note: If your algorithm will not work on the entire `List`, the end condition will be different than this.

Now the next question is, "What should this end condition return?"

A key here is that the function signature states that it returns an `Int`. Therefore, you know that this end condition must return an `Int` of some sort. But what `Int`? Because this is a "sum" algorithm, you also know that you don't want to return anything that will affect the sum. Hmmm … what `Int` can you return when the `Nil` element is reached that won't affect the sum?

The answer is `0`.

(More on this shortly.)

Given that answer, I can update the first `case` condition:

```
def sum(list: List[Int]): Int = list match {
    case Nil => 0
    case ???
}
```

That condition states that if the function receives an empty `List` — denoted by `Nil` — the function will return `0`.

Now we're ready for the third step.

> I'll expand more on the point of returning `0` in this algorithm in the coming lessons, but for now it may help to know that there's a mathematical theory involved in this decision. What's happening here is that you're returning something known as an "identity" element for the current data set and algorithm. As a quick demonstration of what I'm talking about, here are a few other *identity* elements for different data sets and algorithms:
>
> 1) Imagine that you want to write a "product" algorithm for a list of integers. What would you return for the end condition in this case? The correct answer is `1`. This is because the product involves multiplying all elements of the list, and multiplying any number by `1` gives you the original number, so this doesn't affect the final result in any way.
>
> 2) Imagine that you're writing a concatenation algorithm for a `List[String]`. What would you return for the end condition in this case? The correct answer is ` ``'' `, an empty `String` (because once again, it does not affect the final result).

Step 3: What is the algorithm?

Now that you've defined the function signature and the end condition, the final question is, "What is the algorithm at hand?"

When your algorithm will operate on all of the elements in a `List` and the first `case` condition handles the "empty list" case, this question becomes, "What should my function do when it receives a *non-empty* `List`?"

The answer for a "sum" function is that it should add all of the elements in the list. (Similarly, the answer for a "product" algorithm is that it should multiply all of the list elements.)

The sum algorithm

At this point I go back to the original statement of the `sum` algorithm:

> "The sum of a list of integers is the sum of the *head* element, plus the sum of the *tail* elements."

Because the first `case` expression handles the "empty list" case, you know that the second `case` condition should handle the case of the non-empty list. A common way to write the *pattern* for this `case` expression is this:

```
case head :: tail => ???
```

This pattern says, "`head` will be bound to the value of the first element in the `List`, and `tail` will contain all of the remaining elements in the `List`."

Because my description of the algorithm states that the sum is "the sum of the *head* element, plus the sum of the *tail* elements," I start to write a `case` expression, starting by adding the head element:

```
case head :: tail => head + ???
```

and then I write this code to represent "the sum of the tail elements":

```
case head :: tail => head + sum(tail)
```

That is a Scala/FP recursive way of expressing the thought, "The sum of a list of integers is the sum of the *head* element, plus the sum of the *tail* elements."

(I described that thought process in detail in the previous lessons, so I won't repeat all of that thought process here.)

Now that we have the function signature, the end condition, and the main algorithm, we have the completed function:

```
def sum(list: List[Int]): Int = list match {
    case Nil => 0
    case head :: tail => head + sum(tail)
}
```

Naming conventions

As I noted in the previous lessons, when FP developers work with lists, they often prefer to use the variable name `x` to refer to a single element and `xs` to refer to multiple elements, so this function is more commonly written with these variable names:

```
def sum(list: List[Int]): Int = list match {
    case Nil => 0
    case x :: xs => x + sum(xs)
}
```

(But you don't have to use those names; use whatever is easiest for you to read.)

The last two steps are iterative

In practice, the first step — sketching the function signature — is almost always the first step in the process. As I mentioned, it's hard to write a function if you don't know what the inputs and output will be.

But the last two steps — defining the end condition, and writing the algorithm — are interchangeable, and even iterative. For instance, if you're working on a `List` and you want to do something for *every* element in the list, you know the end condition will occur when you reach the `Nil` element. But if you're not going to operate on the entire list, or if you're working with something other than a `List`, it can help to bounce back and forth between the end case and the main algorithm until you come to the solution.

> Note that the `sum` algorithm I've shown specifically works on a Scala `List`, which ends with a `Nil` element. It will not work with other sequences like `Vector`, `ArrayBuffer`, `ListBuffer`, or other sequences that do not have a `Nil` value as the last element in the sequence. I discuss the handling of those other sequences later in the book.

Summary

When I sit down to write a recursive function, I generally think of three things:

- What is the function signature?
- What is the end condition for this algorithm?
- What is the main algorithm?

To solve the problem I almost always write the function signature first, and after that I usually write the end condition next, though the last two steps can also be an iterative process.

What's next

Now that you've seen this "general pattern" of writing recursive functions, the next two lessons are exercises that give you a taste of how to use the patterns to write your own recursive functions.

First, I'll have you write another recursive function to operate on all of the elements in a `List`, and then you'll work on a recursive algorithm that operates on only a subset of a `List`.

37

JVM Stacks and Stack Frames

For functions without deep levels of recursion, there's nothing wrong with the algorithms shown in the previous lessons. I use this simple, basic form of recursion when I know that I'm working with limited data sets. But in applications where you don't know how much data you might be processing, it's important that your recursive algorithms are *tail-recursive*, otherwise you'll get a nasty `StackOverflowError`.

For instance, if you run the `sum` function from the previous lessons with a larger list, like this:

```
object RecursiveSum extends App {

    def sum(list: List[Int]): Int = list match {
        case Nil => 0
        case x :: xs => x + sum(xs)
    }

    val list = List.range(1, 10000)  // MUCH MORE DATA
    val x = sum(list)
    println(x)

}
```

you'll get a `StackOverflowError`, which is *really* counter to our desire to write great, bulletproof, functional programs.

> The actual number of integers in a list needed to produce a `StackOverflowError` with this function will depend on the java command-line settings you use, but the last time I checked the default Java stack size it was 1,024 kb — yes, 1,024 *kilobytes* — just over one million *bytes*. That's

> not much RAM to work with. I write more about this at the end of this lesson, including how to change the default stack size with the `java` command's `-Xss` parameter.

I'll cover tail recursion in the next lesson, but in this lesson I want to discuss the JVM stack and stack frames. If you're not already familiar with these concepts, this discussion will help you understand what's happening here. It can also help you debug "stack traces" in general.

> If you're already comfortable with the JVM stack and stack frames, feel free to skip on to the next lesson.

What is a "Stack"?

To understand the potential "stack overflow" problem of recursive algorithms, you need to understand what happens when you write recursive algorithms.

The first thing to know is that in all computer programming languages there is this thing called "the stack," also known as the "call stack."

Official Java/JVM "stack" definition

Oracle provides the following description of the stack and stack frames as they relate to the JVM:

> "Each JVM thread has a private Java virtual machine stack, created at the same time as the thread. A JVM stack stores frames, also called "stack frames". A JVM stack is analogous to the stack of a conventional language such as C — it holds local variables and partial results, and plays a part in method invocation and return."

Therefore, you can visualize that a single stack has a pile of stack frames that look like Figure 37.1.

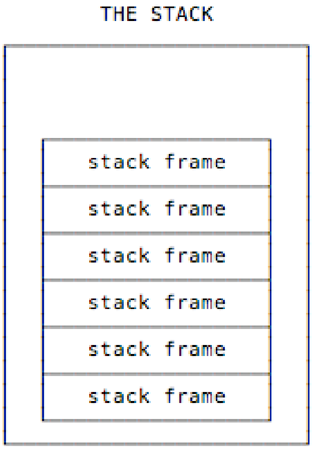


Figure 37.1: A single stack has a pile of stack frames.

As that quote mentions, each thread has its own stack, so in a multi-threaded application there are multiple stacks, and each stack has its own stack of frames, as shown in Figure 37.2.

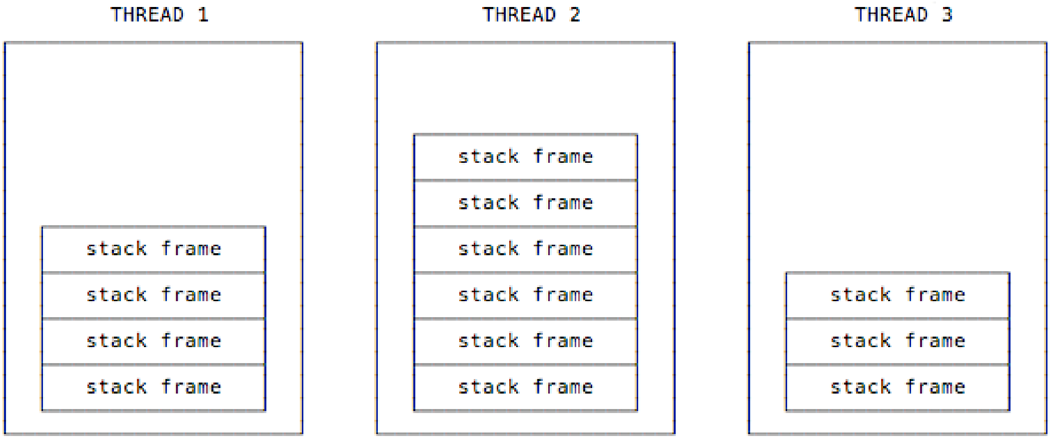


Figure 37.2: Each thread has its own stack.

The Java stack

To explain the stack a little more, all of the following quoted text comes from the free, online version of a book titled, Inside the Java Virtual Machine, by Bill Venners. (I edited the text slightly to include only the portions relevant to stacks and stack frames.)

"When a new thread is launched, the JVM creates a new stack for the thread. A Java stack stores a thread's state in discrete frames. *The JVM only performs two operations directly on Java stacks: it pushes and pops frames.*"

"The method that is currently being executed by a thread is the thread's current method. The stack frame for the current method is the current frame. The class in which the current method is defined is called the current class, and the current class's constant pool is the current constant pool. As it executes a method, the JVM keeps track of the current class and current constant pool. When the JVM encounters instructions that operate on data stored in the stack frame, it performs those operations on the current frame."

"*When a thread invokes a Java method, the JVM creates and pushes a new frame onto the thread's stack.* This new frame then becomes the current frame. As the method executes, it uses the frame to store parameters, local variables, intermediate computations, and other data."

> As the previous paragraph implies, each instance of a method has its own stack frame. Therefore, when you see the term "stack frame," you can think, "all of the stuff a method instance needs."

What is a "Stack Frame"?

The same chapter in that book describes the "stack frame" as follows:

"The stack frame has three parts: local variables, operand stack, and frame data."

You can visualize that as shown in Figure 37.3.

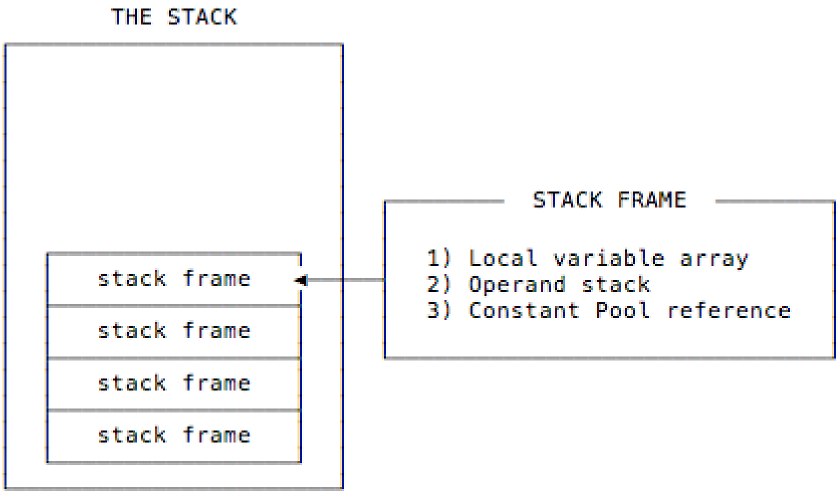


Figure 37.3: Each stack frame has three parts

The book continues:

"The sizes of the local variables and operand stack, which are measured in words, depend upon the needs of each individual method. These sizes are determined at compile time and included in the class file data for each method."

That's important: *the size of a stack frame varies depending on the local variables and operand stack.* The book describes that size like this:

"When the JVM invokes a method, it checks the class data to determine the number of words required by the method in the local variables and operand stack. It creates a stack frame of the proper size for the method and pushes it onto the stack."

Word size, operand stack, and constant pool

These descriptions introduce the phrases word size, operand stack, and constant pool. Here are definitions of those terms:

First, word size is a unit of measure. From Chapter 5 of the same book, the word size can vary in JVM implementations, but it must be at least 32 bits so it can hold a value of type long or double.

Next, the operand stack is defined here on oracle.com, but as a word of warning, that definition gets into machine code very quickly. For instance, it shows how two integers are added together with the iadd instruction. You are welcome to dig into those details, but for our purposes, a simple way to think about the operand stack is that it's memory (RAM) that is used as a working area inside a stack frame.

The Java Run-Time Constant Pool is defined at this oracle.com page, which states, "A run-time constant pool … contains several kinds of constants, ranging from numeric literals known at compile-time, to method and field references that must be resolved at run-time. The run-time constant pool serves a function similar to that of a symbol table for a conventional programming language, although it contains a wider range of data than a typical symbol table."

Summary to this point

I can summarize what we've learned about stacks and stack frames like this:

- Each JVM thread has a private stack, created at the same time as the thread.
- A stack stores frames, also called "stack frames."
- A stack frame is created every time a new method is called.

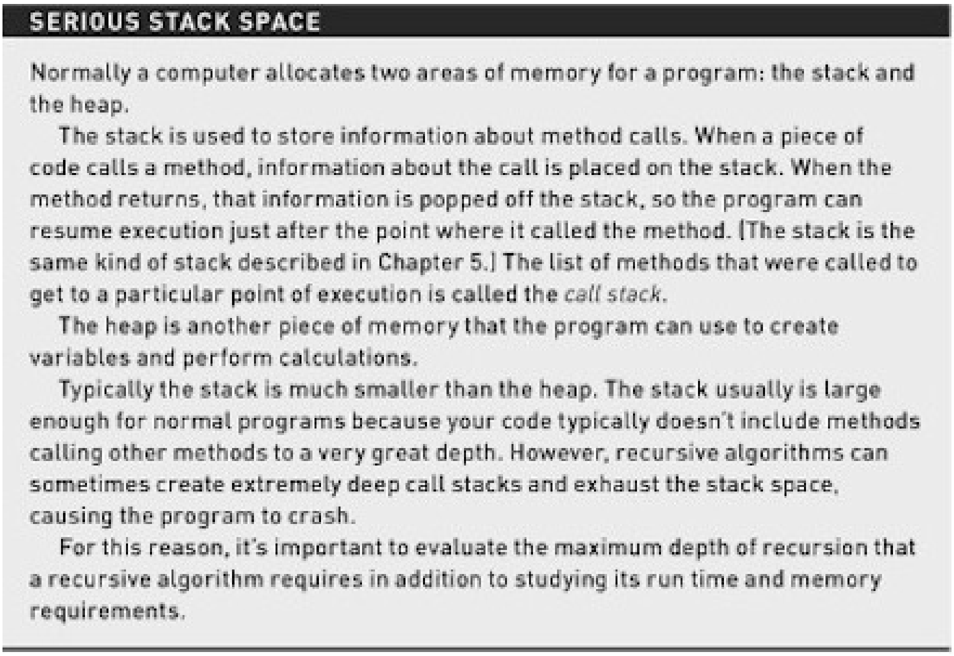
We can also say this about what happens when a Java/Scala/JVM method is invoked:

- When a method is invoked, a new stack frame is created to contain information about that method.
- Stack frames can have different sizes, depending on the method's parameters, local variables, and algorithm.
- As the method is executed, the code can only access the values in the current stack frame, which you can visualize as being the top-most stack frame.

As it relates to recursion, that last point is important. As a function like our `sum` function works on a list, such as `List(1,2,3)`, information about that instance of `sum` is in the top-most stack frame, and that instance of sum can't see the data of other instances of the sum function. This is how what appears to be a single, local variable — like the values head and tail inside of sum — can seemingly have many different values at the same time.

One last resource on the stack and recursion

Not to belabor the point, but I want to share one last description of the stack (and the heap) that has specific comments about recursion. The discussion in Figure 37.4 comes from a book named Algorithms, by Sedgewick and Wayne.



SERIOUS STACK SPACE

Normally a computer allocates two areas of memory for a program: the stack and the heap.

The stack is used to store information about method calls. When a piece of code calls a method, information about the call is placed on the stack. When the method returns, that information is popped off the stack, so the program can resume execution just after the point where it called the method. [The stack is the same kind of stack described in Chapter 5.] The list of methods that were called to get to a particular point of execution is called the *call stack*.

The heap is another piece of memory that the program can use to create variables and perform calculations.

Typically the stack is much smaller than the heap. The stack usually is large enough for normal programs because your code typically doesn't include methods calling other methods to a very great depth. However, recursive algorithms can sometimes create extremely deep call stacks and exhaust the stack space, causing the program to crash.

For this reason, it's important to evaluate the maximum depth of recursion that a recursive algorithm requires in addition to studying its run time and memory requirements.

Figure 37.4: A discussion of the JVM stack and heap

There are two important lines in this description that relate to recursive algorithms:

- "When the method returns, that information is popped off the stack, so the program can resume execution just after the point where it called the method."
- "recursive algorithms can sometimes create extremely deep call stacks and exhaust the stack space."

Analysis

From all of these discussions I hope you can see the potential problem of recursive algorithms:

- When a recursive function calls itself, information for the new instance of the function is pushed onto the stack.
- Each time the function calls itself, another copy of the function information is pushed onto the stack. Because of this, a new stack frame is needed for each level in the recursion.
- As a result, more and more memory that is allocated to the stack is consumed as the function recurses. If the `sum` function calls itself a million times, a million stack frames are created.

38

A Visual Look at Stacks and Frames

Given the background information of the previous lesson, let's take a visual look at how the JVM stack and stack frames work by going back to our recursive sum function from the previous lesson.

Before the `sum` function is initially called, the only thing on the call stack is the application's `main` method, as shown in Figure 38.1.

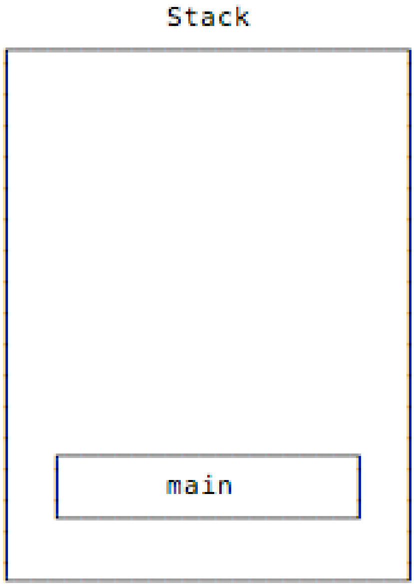


Figure 38.1: `main` *is the only thing on the call stack before* `sum` *is called.*

Then `main` calls `sum` with `List(1,2,3)`, which I show in Figure 38.2 without the "List" to keep things simple.

The data that's given to `sum` matches its second `case` expression, and in my pseudocode, that expression evaluates to this:

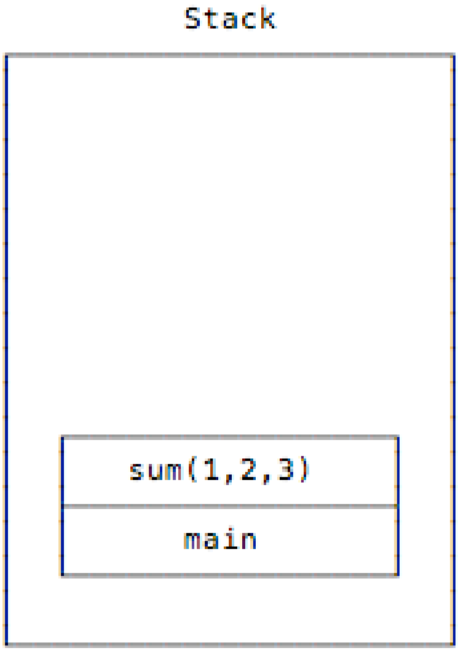


Figure 38.2: The first `sum` *call is added to the stack.*

```
return 1 + sum(2,3)
```

Next, when a new instance of `sum` is called with `List(2,3)`, the stack looks as shown in Figure 38.3.

Again the second `case` expression is matched inside of `sum`, and it evaluates to this:

```
return 2 + sum(3)
```

When a new instance of `sum` is called with the input parameter `List(3)`, the stack looks like Figure 38.4.

Again the second `case` expression is matched, and that code evaluates to this:

```
return 3 + sum(Nil)
```

Finally, another instance of `sum` is called with the input parameter `List()` — also known as `Nil` — and the stack now looks like Figure 38.5.

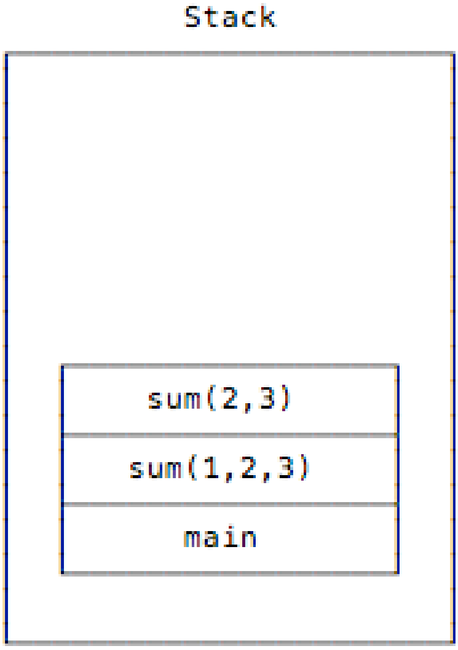


Figure 38.3: The second `sum` *call is added to the stack.*

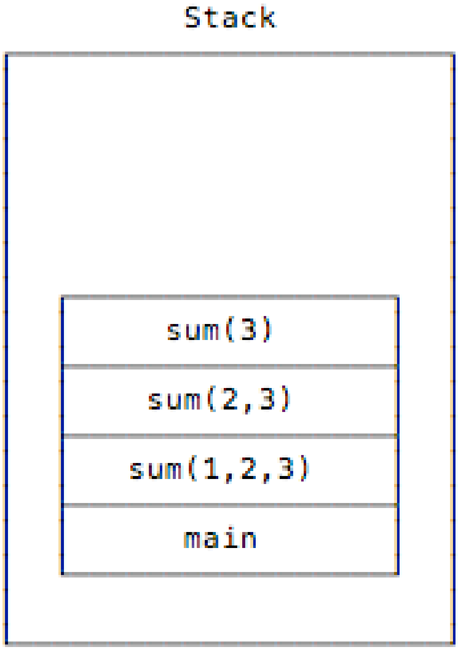


Figure 38.4: The third `sum` *call is added to the stack.*

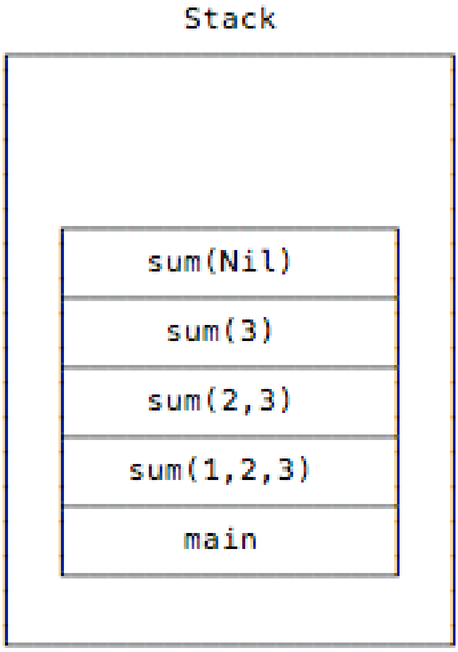


Figure 38.5: The final `sum` *call is added to the stack.*

This time, when `sum(Nil)` is called, the first `case` expression is matched:

```
case Nil => 0
```

That pattern match causes this `sum` instance to return `0`, and when it does, the call stack unwinds and the stack frames are popped off of the stack, as shown in the series of images in Figure 38.6.

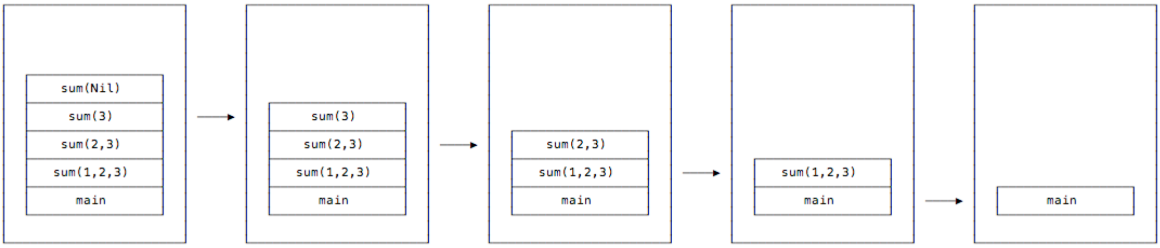


Figure 38.6: The unwinding of the call stack.

In this process, as each `sum` call returns, its frame is popped off of the stack, and when the recursion completely ends, the main method is the only frame left on the call stack. (The value `6` is also returned by the first `sum` invocation to the place where it was called in the `main` method.)

I hope that gives you a good idea of how recursive function calls are pushed-on and popped-off the JVM call stack.

Manually dumping the stack with the sum example

If you want to explore this in code, you can also see the series of `sum` stack calls by modifying the `sum` function. To do this, add a couple of lines of code to the `Nil` case to print out stack trace information when that `case` is reached:

```
def sum(list: List[Int]): Int = list match {
    case Nil => {
        // this manually creates a stack trace
        val stackTraceAsArray = Thread.currentThread.getStackTrace
        stackTraceAsArray.foreach(println)
        // return 0 as before
        0
    }
    case x :: xs => x + sum(xs)
}
```

Now, if you call `sum` with a list that goes from 1 to 5:

```
val list = List.range(1, 5)
sum(list)
```

you'll get this output when the `Nil` case is reached:

```
java.lang.Thread.getStackTrace(Thread.java:1588)
recursion.SumWithStackDump$.sum(SumWithStackDump.scala:19)
recursion.SumWithStackDump$.sum(SumWithStackDump.scala:19)
recursion.SumWithStackDump$.sum(SumWithStackDump.scala:19)
recursion.SumWithStackDump$.sum(SumWithStackDump.scala:19)
recursion.SumWithStackDump$.sum(SumWithStackDump.scala:19)
```

While that output isn't too exciting, it shows that when the stack dump is manually triggered when the `Nil` case is reached, the `sum` function is on the stack five times. You can verify that this is correct by repeating the test with a `List` that has three elements, in which case you'll see the `sum` function referenced only three times in the output:

```
java.lang.Thread.getStackTrace(Thread.java:1588)
recursion.SumWithStackDump$.sum(SumWithStackDump.scala:13)
recursion.SumWithStackDump$.sum(SumWithStackDump.scala:19)
recursion.SumWithStackDump$.sum(SumWithStackDump.scala:19)
```

Clearly the `sum` function is being added to the stack over and over again, once for each call.

> I encourage you to try this on your own to become comfortable with what's happening.

Summary: Our current problem with "basic recursion"

I hope this little dive into the JVM stack and stack frames helps to explain our current problem with "basic recursion." As mentioned, if I try to pass a `List` with 10,000 elements into the current recursive `sum` function, it will generate a `StackOverflowError`. Because we're trying to write bulletproof programs, this isn't good.

What's next

Now that we looked at (a) basic recursion with the `sum` function, (b) how that works with stacks and stack frames in the last two lessons, and (c) how basic recursion can throw a `StackOverflowError` with large data sets, the next lesson shows how to fix these problems with something called "tail recursion."

See also

I didn't get into all of the nitty-gritty details about the stack and stack frames in this lesson. If you want to learn more about the stack, here are some excellent resources:

- Chapter 5 of Inside the Java Virtual Machine, by Bill Venners is an excellent resource. You may not need to read anything more than the content at this URL.
- Chapter 2 of Oracle's JVM Specification is also an excellent resource.
- This article titled, Understanding JVM Internals on cubrid.org is another good read.
- If you want even more gory details, an article titled, Understanding the Stack on umd.edu is excellent.
- Here's an article I wrote about the differences between the stack and the heap a long time ago.

One more thing: Viewing and setting the JVM stack size

"Well," you say, "these days computers have crazy amounts of memory. Why is this such a problem?"

According to this Oracle document, with Java 6 the default stack size was very low: 1,024k on both Linux and Windows.

I encourage you to check the JVM stack size on your favorite computing platform(s). One way to check it is with a command like this on a Unix-based system:

```
java -XX:+PrintFlagsFinal -version | grep -i stack
```

When I do this on my current Mac OS X system, I see that the `ThreadStackSize` is `1024`. I dug through this oracle.com documentation to find that this "1024" means "1,024 Kbytes".

It's important to know that you can also control the JVM stack size with the `-Xss` command line option:

```
$ java -Xss 1M ... (the rest of your command line here)
```

That command sets the stack size to one megabyte. You specify the memory size attribute as `m` or `M` after the numeric value to get megabytes, as in `1m` or `1M` for one megabyte.

> Use `g` or `G` to specify the size in gigabytes, but if you're trying to use many MB or GB for the stack size, you're doing something wrong. You may need this gigabytes option for the `Xmx` option, but you should never need it for this `Xss` attribute.

The `Xss` option can be helpful if you run into a `StackOverflowError` — although the next lesson on *tail recursion* is intended to help you from ever needing this command line option.

More JVM memory settings

As a final note, you can find more options for controlling Java application memory use by looking at the output of the `java -X` command:

```
$ java -X
```

If you dig through the output of that command, you'll find that the command-line arguments specifically related to Java application memory use are:

```
-Xms  set initial Java heap size
-Xmx  set maximum Java heap size
-Xss  set java thread stack size
```

You can use these parameters on the java command line like this:

```
java -Xms64m -Xmx1G myapp.jar
```

As before, valid memory values end with `m` or `M` for megabytes, and `g` or `G` for gigabytes:

```
-Xms64m or -Xms64M
-Xmx1g or -Xmx1G
```

39

Tail-Recursive Algorithms

> *"Tail recursion is its own reward."*
>
> From the "Functional" cartoon on xkcd.com.

Goals

The main goal of this lesson is to solve the problem shown in the previous lessons: Simple recursion creates a series of stack frames, and for algorithms that require deep levels of recursion, this creates a `StackOverflowError` (and crashes your program).

"Tail recursion" to the rescue

Although the previous lesson showed that algorithms with deep levels of recursion can crash with a `StackOverflowError`, all is not lost. With Scala you can work around this problem by making sure that your recursive functions are written in a *tail-recursive* style.

A tail-recursive function is just a function whose *very last action* is a call to itself. When you write your recursive function in this way, the Scala compiler can optimize the resulting JVM bytecode so that the function requires only one stack frame — as opposed to one stack frame for each level of recursion!

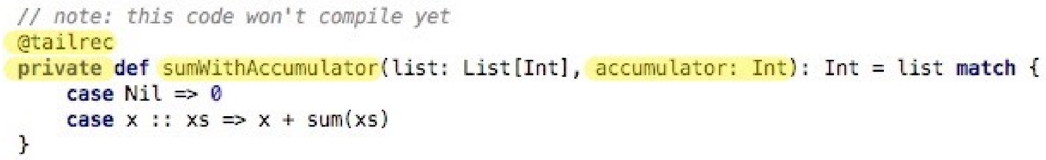
On Stack Overflow, Martin Odersky explains tail-recursion in Scala:

> "Functions which call themselves as their last action are called tail-recursive. The Scala compiler detects tail recursion and replaces it with a jump back to the beginning of the function, after updating the

> function parameters with the new values … as long as the last thing you do is calling yourself, it's automatically tail-recursive (i.e., optimized)."

But that `sum` function looks tail-recursive to me …

"Hmm," you might say, "if I understand Mr. Odersky's quote, the `sum` function you wrote at the end of the last lesson (shown in Figure 39.1) sure looks tail-recursive to me."



```
// note: this code won't compile yet
@tailrec
private def sumWithAccumulator(list: List[Int], accumulator: Int): Int = list match {
    case Nil => 0
    case x :: xs => x + sum(xs)
}
```

Figure 39.1: The call to `sum` *appears to be the last action.*

"Isn't the 'last action' a call to itself, making it tail-recursive?"

If that's what you're thinking, fear not, that's an easy mistake to make. But the answer is no, this function is not tail-recursive. Although `sum(tail)` is at the end of the second `case` expression, you have to think like a compiler here, and when you do that you'll see that the last two actions of this function are:

1. Call `sum(xs)`
2. When that function call returns, add its value to `x` and return that result

When I make that code more explicit and write it as a series of one-line statements, you see that it looks like this:

```
val s = sum(xs)
val result = x + s
return result
```

As shown, the last calculation that happens before the `return` statement is that the sum of `x` and `s` is calculated. If you're not 100% sure that you believe that, there are a few ways you can prove it to yourself.

1) Proving it with the previous "stack trace" example

One way to "prove" that the `sum` algorithm is not tail-recursive is with the "stack trace" output from the previous lesson. The JVM output shows the `sum` method is called once for each step in the recursion, so it's clear that the JVM feels the need to create a new instance of `sum` for each element in the collection.

2) Proving it with the `@tailrec` *annotation*

A second way to prove that `sum` isn't tail-recursive is to attempt to tag the function with a Scala annotation named `@tailrec`. This annotation won't compile unless the function is tail-recursive. (More on this later in this lesson.)

If you attempt to add the `@tailrec` annotation to `sum`, like this:

```
// need to import tailrec before using it
import scala.annotation.tailrec

@tailrec
def sum(list: List[Int]): Int = list match {
    case Nil => 0
    case x :: xs => x + sum(xs)
}
```

the `scalac` compiler (or your IDE) will show an error message like this:

```
Sum.scala:10: error: could not optimize @tailrec annotated method sum:
it contains a recursive call not in tail position
def sum(list: List[Int]): Int = list match {
                                ^
```

This is another way to "prove" that the Scala compiler doesn't think `sum` is tail-recursive.

> Note: The text, "it contains a recursive call not in tail position," is the Scala error message you'll see whenever a function tagged with `@tailrec` isn't really tail-recursive.

So, how do I write a tail-recursive function?

Now that you know the current approach isn't tail-recursive, the question becomes, "How do I make it tail-recursive?"

A common pattern used to make a recursive function that "accumulates a result" into a tail-recursive function is to follow a series of simple steps:

1. Keep the original function signature the same (i.e., `sum`'s signature).
2. Create a second function by (a) copying the original function, (b) giving it a new name, (c) making it `private`, (d) giving it a new "accumulator" input parameter, and (e) adding the `@tailrec` annotation to it.
3. Modify the second function's algorithm so it uses the new accumulator. (More on this shortly.)
4. Call the second function from inside the first function. When you do this you give the second function's accumulator parameter a "seed" value (a little like the *identity* value I wrote about in the previous lessons).

Let's jump into an example to see how this works.

Example: How to make `sum` tail-recursive

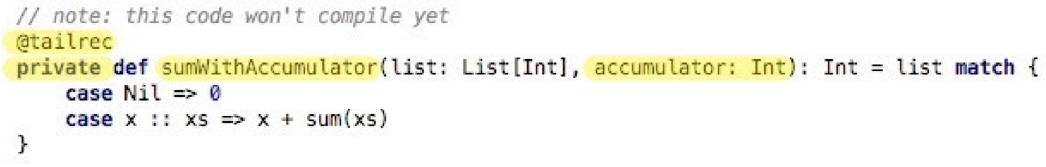
1) Leave the original function signature the same

To begin the process of converting the recursive `sum` function into a *tail-recursive* `sum` algorithm, leave the external signature of `sum` the same as it was before:

```
def sum(list: List[Int]): Int = ...
```

2) Create a second function

Now create the second function by copying the first function, giving it a new name, marking it `private`, giving it a new “accumulator” parameter, and adding the `@tailrec` annotation to it. The highlights in Figure 39.2 show the changes.



```
// note: this code won't compile yet
@tailrec
private def sumWithAccumulator(list: List[Int], accumulator: Int): Int = list match {
    case Nil => 0
    case x :: xs => x + sum(xs)
}
```

Figure 39.2: Starting to create the second function.

This code won’t compile as shown, so I’ll fix that next.

> Before moving on, notice that the data type for the accumulator (`Int`) is the same as the data type held in the `List` that we’re iterating over.

3) Modify the second function’s algorithm

The third step is to modify the algorithm of the newly-created function to use the accumulator parameter. The easiest way to explain this is to show the code for the solution, and then explain the changes. Here’s the source code:

```
@tailrec
private def sumWithAccumulator(list: List[Int], accumulator: Int): Int = {
    list match {
        case Nil => accumulator
        case x :: xs => sumWithAccumulator(xs, accumulator + x)
    }
}
```

Here’s a description of how that code works:

- I marked it with `@tailrec` so the compiler can help me by verifying that my code truly is tail-recursive.
- `sumWithAccumulator` takes two parameters, `list: List[Int]`, and `accumulator: Int`.
- The first parameter is the same list that the `sum` function receives.
- The second parameter is new. It's the "accumulator" that I mentioned earlier.
- The inside of the `sumWithAccumulator` function looks similar. It uses the same match/case approach that the original `sum` method used.
- Rather than returning `0`, the first `case` statement returns the `accumulator` value when the `Nil` pattern is matched. (More on this shortly.)
- The second `case` expression is tail-recursive. When this case is matched it immediately calls `sumWithAccumulator`, passing in the `xs` (tail) portion of `list`. What's different here is that the second parameter is the sum of the `accumulator` and the head of the current list, `x`.
- Where the original `sum` method passed itself the tail of `xs` and then later added that result to `x`, this new approach keeps track of the accumulator (total sum) value as each recursive call is made.

The result of this approach is that the "last action" of the `sumWithAccumulator` function is this call:

```
sumWithAccumulator(xs, accumulator + x)
```

Because this last action really is a call back to the same function, the JVM can optimize this code as Mr. Odersky described earlier.

4) Call the second function from the first function

The fourth step in the process is to modify the original function to call the new function. Here's the source code for the new version of `sum`:

```
def sum(list: List[Int]): Int = sumWithAccumulator(list, 0)
```

Here's a description of how it works:

- The `sum` function signature is the same as before. It accepts a `List[Int]` and returns an `Int` value.
- The body of `sum` is just a call to the `sumWithAccumulator` function. It passes the original `list` to that function, and also gives its `accumulator` parameter an initial seed value of `0`.

Note that this "seed" value is the same as the *identity* value I wrote about in the previous recursion lessons. In those lessons I noted:

- The identity value for a sum algorithm is 0.
- The identity value for a product algorithm is 1.
- The identity value for a string concatenation algorithm is `""`.

A few notes about `sum`

Looking at `sum` again:

```
def sum(list: List[Int]): Int = sumWithAccumulator(list, 0)
```

a few key points about it are:

- Other programmers will call `sum`. It's the "Public API" portion of the solution.
- It has the same function signature as the previous version of `sum`. The benefit of this is that other programmers won't have to provide the initial seed value. In fact, they won't know that the internal algorithm uses a seed value. All they'll see is `sum`'s signature:

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

A slightly better way to write `sum`

Tail-recursive algorithms that use accumulators are typically written in the manner shown, with one exception: Rather than mark the new accumulator function as `private`, most Scala/FP developers like to put that function *inside* the original function as a way to limit its scope.

> When doing this, the thought process is, "Don't expose the scope of `sumWithAccumulator` unless you want other functions to call it."

When you make this change, the final code looks like this:

```
// tail-recursive solution
def sum(list: List[Int]): Int = {
    @tailrec
    def sumWithAccumulator(list: List[Int], currentSum: Int): Int = {
        list match {
            case Nil => currentSum
            case x :: xs => sumWithAccumulator(xs, currentSum + x)
        }
    }
    sumWithAccumulator(list, 0)
}
```

Feel free to use either approach. (Don't tell anyone, but I prefer the first approach; I think it reads more easily.)

A note on variable names

If you don't like the name `accumulator` for the new parameter, it may help to see the function with a different name. For a "sum" algorithm a name like `runningTotal` or `currentSum` may be more meaningful:

```
// tail-recursive solution
def sum(list: List[Int]): Int = {
    @tailrec
    def sumWithAccumulator(list: List[Int], currentSum: Int): Int = {
        list match {
            case Nil => currentSum
            case x :: xs => sumWithAccumulator(xs, currentSum + x)
        }
    }
    sumWithAccumulator(list, 0)
}
```

I encourage you to use whatever name makes sense to you. Personally I prefer `currentSum` for this algorithm, but you'll often hear this approach referred to as using an "accumulator," which is why I used that name first.

> Of course you can also name the inner function whatever you'd like to call it.

Proving that this is tail-recursive

Now let's prove that the compiler thinks this code is tail-recursive.

First proof

The first proof is already in the code. When you compile this code with the `@tailrec` annotation and the compiler doesn't complain, you know that the compiler believes the code is tail-recursive.

Second proof

If for some reason you don't believe the compiler, a second way to prove this is to add some debug code to the new `sum` function, just like we did in the previous lessons. Here's the source code for a full Scala `App` that shows this approach:

```
import scala.annotation.tailrec

object SumTailRecursive extends App {

    // call sum
    println(sum(List.range(1, 10)))

    // the tail-recursive version of sum
    def sum(list: List[Int]): Int = {
        @tailrec
        def sumWithAccumulator(list: List[Int], currentSum: Int): Int = {
            list match {
               case Nil => {
                val stackTraceAsArray = Thread.currentThread.getStackTrace
                    stackTraceAsArray.foreach(println)
                    currentSum
                }
                case x :: xs => sumWithAccumulator(xs, currentSum + x)
            }
        }
        sumWithAccumulator(list, 0)
    }

}
```

> Note: You can find this code at this Github link. This code includes a few ScalaTest tests, including one test with a `List` of 100,000 integers.

When I compile that code with `scalac`:

```
$ scalac SumTailRecursive.scala
```

and then run it like this:

```
$ scala SumTailRecursive
```

I get a lot of output, but if I narrow that output down to just the `sum`-related code, I see this:

```
[info] Running recursion.TailRecursiveSum
java.lang.Thread.getStackTrace(Thread.java:1552)
recursion.TailRecursiveSum$.sumWithAccumulator$1(TailRecursiveSum.scala:16)
recursion.TailRecursiveSum$.sum(TailRecursiveSum.scala:23)
//
// lots of other stuff here ...
//
scala.App$class.main(App.scala:76)
recursion.TailRecursiveSum$.main(TailRecursiveSum.scala:5)
recursion.TailRecursiveSum.main(TailRecursiveSum.scala)
45
```

As you can see, although the `List` in the code has 10 elements, there's only one call to `sum`, and more importantly in this case, only one call to `sumAccumulator`. You can now safely call `sum` with a list that has 10,000 elements, 100,000 elements, etc., and it will work just fine without blowing the stack. (Go ahead and test it!)

> Note: The upper limit of a Scala `Int` is `2,147,483,647`, so at some point you'll create a number that's too large for that. Fortunately a `Long` goes to `2^63-1` (which is `9,223,372,036,854,775,807`), so that problem is easily remedied. (If that's not big enough, use a `BigInt`.)

Summary

In this lesson I:

- Defined tail recursion
- Introduced the `@tailrec` annotation
- Showed how to write a tail-recursive function
- Showed a formula you can use to convert a simple recursive function to a tail-recursive function

What's next

This lesson covered the basics of converting a simple recursive function into a tail-recursive function. I'm usually not smart enough to write a tail-recursive function right away, so I usually write my algorithms using simple recursion, then convert them to use tail-recursion.

To help in your efforts, the next lesson will show more examples of tail-recursive for different types of algorithms.

See also

- My list of Scala recursion examples
- Martin Odersky explaining tail recursion on Stack Overflow

40

A First Look at "State"

In the next lesson I'm going to start writing a little command-line game, but before I get into that I want to discuss the general concept of handling "state" in software applications.

Every non-trivial application maintains some sort of *state*. For instance, the state of a word processing application is the current document, along with whether the document has been saved or not (whether the document is "clean" or "dirty"). Similarly, the state of a spreadsheet application is the spreadsheet and its clean/dirty state. Web versions of these applications have additional state, such as who the current user is, when they logged in, what their IP address is, etc.

Even voice applications like Siri and Amazon Echo have state. As I learned in writing SARAH, one thing you need to do is to maintain speaking/listening state, otherwise the computer will hear itself talking, then respond to itself, eventually kicking off an endless loop.

Siri and others are also gaining a concept that I call *context*, or the "context of a conversation," which also requires state management. Imagine asking Siri to order a pizza. It will respond by asking what toppings you want, where you want to order the pizza from, how you want to pay, etc. This is "conversational state."

Handling state in a game

In my spare time I work on developing an Android football game where I play against a computer opponent. If you know American Football (as opposed to what we Americans call "soccer"), in between each play you can think of the state of a football game as having these attributes:

- Which team has the ball (you are on offense or defense)

- Current field position
- Down and distance (such as "1st and 10")
- Current score
- Time remaining

> There are more state variables than this, but I'm keeping this example simple.

In Scala you might model this game state like this:

```
case class GameState (
    iHaveTheBall: Boolean,
    fieldPosition: Int,
    down: Int,
    distance: Int,
    myScore: Int,
    computerScore: Int,
    timeRemaining: Int
)
```

On the first play of the game the initial state might look like this:

```
GameState (
    iHaveTheBall: true,
    fieldPosition: 25,
    down: 1,
    distance: 10,
    myScore: 0,
    computerScore: 0,
    timeRemaining: 3600
)
```

Then, after the next play the state might look like this:

```
GameState (
    iHaveTheBall: true,
    fieldPosition: 29,
    down: 2,
    distance: 6,
    myScore: 0,
    computerScore: 0,
    timeRemaining: 3536
)
```

A football game typically has about 150 plays, so in my game there is a `GameState` instance for each of those plays.

Why state is important

State is important for many reasons, not the least of which is to know when the game is over and who won. An important part about state in my football game is that I use it to help the computer make decisions about what plays it calls.

When the computer is playing on offense is uses a function that looks like this:

```
val possiblePlays: List[OffensivePlay] =
    OffensiveCoordinator.determinePossiblePlays(gameState)
```

The `determinePossiblePlays` function is a pure function. I pass `GameState` into it, and with thousands of lines of purely functional code behind it, it returns a list of all the possible plays that the algorithms believe make sense for the state that was passed in.

For instance, if it's fourth down and goal at the opponent's one-yard line with five seconds left in the game and the computer is down 21-17, it's smart enough to know that it needs to try to score a touchdown rather than kick a field goal. This is what I mean by "state" in the context of a football game.

> As the game gets smarter I also maintain a history of all previously-called plays, so the computer can adjust its play calls based on the player's tendencies.

More state

As you can imagine, a point of sales application for a pizza store will have state that includes:

- The number and types of pizzas ordered
- Customer contact information
- Customer payment information
- The date and time of the order
- Who took the order
- More …

Once you begin to think about it this way, you'll see that every application maintains state of some sort.

State and functional programming

As I mentioned, my football game has about 150 `GameState` instances for every game. In the context of functional programming, this raises an interesting question: In Scala/FP I can only have `val` instances, so how can I possibly create 150 new variables for each game? Put another way, if you assume that I keep all of the plays in a `List`, the question becomes, "How do I append `GameState` values to an immutable `List`?"

Questions like this bring you to a key point I got to when I was learning FP:

- How am I supposed to handle I/O, which by its very nature is impure?
- How am I supposed to handle state?

In the next lesson I'll show one way to handle state in a simple game by building on what you just learned in the previous lessons: recursion.

41

A Functional Game (With a Little Bit of State)

> *"In theory, theory and practice are the same. In practice, they're not."*
>
> Yogi Berra

Introduction

Now that I've given you a little background about what I think "state" is, let's build a simple game that requires us to use state. I'll build the game using recursion, and also *immutable state* — something I had never heard of when I first starting writing the Scala Cookbook.

Goals

Here are my goals for this lesson:

- To write our first functional application
- Show a first example of how to handle "state" in a Scala/FP application

Source code

The best way to understand this lesson is to have its source code open in an IDE as you read it. The source code is available at this Github URL:

- My "Coin Flip" game

Some of this project's code is a little wide and won't show up well in a PDF format. You'll really want to check the code out of Github to see it properly.

Coin Flip: A simple FP game

To get started using state in a Scala application, I'll build a little game you can play at the command line. The application will flip a coin (a virtual coin), and as the player, your goal is to guess whether the result is heads or tails. The computer will keep track of the total number of flips and the number of correct guesses.

When you start the game, you'll see this command-line prompt:

```
(h)eads, (t)ails, or (q)uit: _
```

This is how the application prompts you for your guess. Enter `h` for heads, `t` for tails, or `q` to quit the game. If you enter `h` or `t`, the application will flip a virtual coin, then let you know if your guess was correct or not.

As an example of how it works, I just played the game and made four guesses, and the input/output of that session looks like this:

```
(h)eads, (t)ails, or (q)uit: h
Flip was Heads. #Flips: 1, #Correct: 1

(h)eads, (t)ails, or (q)uit: h
Flip was Tails. #Flips: 2, #Correct: 1

(h)eads, (t)ails, or (q)uit: h
Flip was Heads. #Flips: 3, #Correct: 2

(h)eads, (t)ails, or (q)uit: t
Flip was Tails. #Flips: 4, #Correct: 3

(h)eads, (t)ails, or (q)uit: q

=== GAME OVER ===
```

```
#Flips: 4, #Correct: 3
```

Admittedly this isn't the most exciting game in the world, but it turns out to be a nice way to learn how to handle immutable state in a Scala/FP application.

One note before proceeding: The input/output in this game will *not* be handled in a functional way. I'll get to that in a future lesson.

On to the game!

Coin Flip game state

Let's analyze how this game works:

- The computer is going to flip a virtual coin.
- You're going to guess whether that result is heads or tails.
- You can play the game for as many flips as you want.
- After each flip the output will look like this:

  Flip was Tails. #Flips: 4, #Correct: 2

These statements tell us a few things about the game state:

- We need to track how many coin flips there are.
- We need to track how many guesses the player made correctly.

I could track more information, such as the history of the guess for each coin flip and the actual value, but to keep it simple, all I want to do at this time is to track (a) the number of flips, and (b) the number of correct guesses. As a result, a first stab at modeling the game state looks like this:

```
case class GameState (
    numFlips: Int,
    numCorrectGuesses: Int
)
```

Game pseudocode

Next, let's start working on the game code.

You know you're going to need some sort of main loop, and in the imperative world, pseudocode for that loop looks like this:

```
var input = ""
while (input != "q") {
    // prompt the player to select heads, tails, or quit
    // get the player's input
    if (input == "q") {
        print the game summary
        quit
    }
    // flip the coin
    // see if the player guessed correctly
    // print the #flips and #correct
}
```

I/O functions

Alas, that's not how I'll write the loop, but it does give me an idea of some I/O functions I'm going to need. From that pseudocode it looks like I'm going to need these functions:

- A "show prompt" function
- A "get user input" function
- A function to print the number of flips and correct answers

These functions have nothing to do with FP — they're impure I/O functions that connect our application to the outside world — so I'll write them in a standard Scala/OOP way. Here's the "show prompt" function:

```
def showPrompt: Unit = { print("\n(h)eads, (t)ails, or (q)uit: ") }
```

Next, here's the "get user input" function:

```
def getUserInput = readLine.trim.toUpperCase
```

Prior to Scala 2.11.0, `readLine` was made available to you without an `import` statement via Scala's `Predef` object, but since then it's available at *scala.io.StdIn.readLine*. Notice that I convert all input to uppercase to make it easier to work with later.

Next, while the game is being played I want to print output like this:

```
Flip was Tails. #Flips: 4, #Correct: 3
```

and when the game is over I want to print this output:

```
=== GAME OVER ===
#Flips: 4, #Correct: 3
```

To accommodate these needs I create these functions:

```
def printableFlipResult(flip: String) = flip match {
    case "H" => "Heads"
    case "T" => "Tails"
}

def printGameState(printableResult: String, gameState: GameState): Unit = {
    print(s"Flip was $printableResult. ")
    printGameState(gameState)
}

def printGameState(gameState: GameState): Unit = {
   println(s"#Flips: ${gameState.numFlips}, #Correct: ${gameState.numCorrect}")
}

def printGameOver: Unit = println("\n=== GAME OVER ===")
```

Note that the `printGameState` functions take the `GameState` as an input parameter, and use its fields to print the output. The assumption is that these functions always receive the latest, up-to-date `GameState` instance.

If you know Scala, that's all fairly standard "print this out" and "read this in" code.

Declaring the `Unit` return type

Note that in these examples I use `: Unit =` syntax on the functions that have no return type. Methods that have a `Unit` return type are called *procedures*, and the Procedure Syntax in the Scala Style Guide recommends declaring the `Unit` return type, so I've shown it here.

Writing a toin coss function

When you look back at this piece of the original pseudocode:

```
// flip the coin
```

you'll see that one more thing I can get out of the way before writing the main loop is a function to simulate a coin toss.

A simple way to simulate a toin coss is to use a random number generator and limit the generator to return values of `0` and `1`, where `0` means "heads" and `1` mean "tails." This is how you limit Scala's `Random.nextInt` method to yield only `0` or `1`:

```
val r = new scala.util.Random
r.nextInt(2)
```

The `r.nextInt(2)` code tells `nextInt` to return integer values that are less than `2`, i.e., `0` and `1`.

Knowing that, I can write a coin flip function like this:

```
// returns "H" for heads, "T" for tails
def tossCoin(r: Random) = {
    val i = r.nextInt(2)
    i match {
        case 0 => "H"
        case 1 => "T"
    }
}
```

> Question: Do you think this is a pure function? If so, why do you think so, and if not, why not?

With these functions out of the way, let's get to the main part of the lesson: how to write the main loop of the program with an immutable game state.

Writing the main loop in FP style

So now we need a "loop" … how can we write one in an FP style? Using the tools we know so far, the best way to handle this is with our new friend, recursion.

Because you may have never done this before, let me add a few important notes:

- With recursion the main loop is going to call itself repeatedly (recursively)
- Because the game state needs to be updated as the game goes along, a `GameState` instance needs to be passed into each recursive call
- Because each instance of the loop will simulate the flip of a coin, and because the `tossCoin` function requires a `scala.util.Random` instance, it's also best to pass a `Random` instance into each recursive call as well

Given that background, I can start writing some code. First, here's the `GameState` I showed earlier:

```
case class GameState (
    numFlips: Int,
    numCorrectGuesses: Int
)
```

Next, I know I'm going to need (a) a Scala `App`, (b) initial `GameState` and `Random` instances, and (c) some sort of `mainLoop` call to get things started. I also know that `mainLoop` will take the `GameState` and `Random` instances, which leads me to this code:

```
object CoinFlip extends App {
    val s = GameState(0, 0)
    val r = new Random
    mainLoop(s, r)
}
```

Next, I can sketch the `mainLoop` function like this:

```
@tailrec
def mainLoop(gameState: GameState, random: Random) {
    // a) prompt the user for input
    // b) get the user's input
    // c) flip the coin
    // d) compare the flip result to the user's input
    // e) write the output
    // f) if the user didn't type 'h', loop again:
    mainLoop(newGameState, random)
}
```

If you feel like you understand what I've sketched in this `mainLoop` code, I encourage you to set this book aside and work on filling out `mainLoop`'s body on your own, using (a) the I/O functions I showed earlier and (b) any other code you might need. That's all that needs to be done now: fill out the body, and figure out where the recursive `mainLoop` call (or calls) need to be made.

Writing the skeleton code

The next thing I did to solve this problem was to stub out the following skeleton code:

```
object CoinFlip extends App {

    val r = Random
    val s = GameState(0, 0)
    mainLoop(s, r)

    @tailrec
    def mainLoop(gameState: GameState, random: Random) {

        // a) prompt the user for input
        showPrompt()

        // b) get the user's input
        val userInput = getUserInput()

        userInput match {
            case "H" | "T" => {
                // c) flip the coin
                val coinTossResult = tossCoin(random)
                val newNumFlips = gameState.numFlips + 1

                // d) compare the flip result to the user's input
                if (userInput == coinTossResult) {
                    // they guessed right
                    // e) write the output
                    // f) if the user didn't type 'h', loop again:
                    mainLoop(newGameState, random)
                } else {
                    // they guessed wrong
                    // e) write the output
                    // f) if the user didn't type 'h', loop again:
```

```
                    mainLoop(newGameState, random)
                }
            }
            case _   => {
                // assume they type 'Q'
                println("\n=== GAME OVER ===")
                printGameState(gameState)
                // we return out of the recursion here
            }
        }
    }

}
```

That code is slightly different than my pseudocode, but it's in the ballpark.

Now all I need to do is finish off the 'e' and 'f' portions of the algorithm. I'll show those sections in the completed code that follows.

The complete source code

The following source code shows the first cut of my solution for this application.

First, I put all of my "utility" functions in a separate object named `CoinFlipUtils`, in a file named *CoinFlipUtils.scala*:

```
package com.alvinalexander.coinflip.v1

import scala.util.Random
import scala.io.StdIn.readLine

object CoinFlipUtils {

    def showPrompt(): Unit = { print("\n(h)eads, (t)ails, or (q)uit: ") }

    def getUserInput(): String = readLine.trim.toUpperCase
```

```
    def printableFlipResult(flip: String): String = flip match {
        case "H" => "Heads"
        case "T" => "Tails"
    }

   def printGameState(printableFlipResult: String, gameState: GameState): Unit = {
        print(s"Flip was $printableFlipResult. ")
        printGameState(gameState)
    }

    def printGameState(gameState: GameState): Unit = {
     println(s"#Flips: ${gameState.numFlips}, #Correct: ${gameState.numCorrect}")
    }

    def printGameOver(): Unit = println("\n=== GAME OVER ===")

    // returns "H" for heads, "T" for tails
    def tossCoin(r: Random): String = {
        val i = r.nextInt(2)
        i match {
            case 0 => "H"
            case 1 => "T"
        }
    }

}
```

I did that to keep the code organized, and also to keep my next file smaller. Here's the source code for *CoinFlip.scala*, which primarily consists of the `mainLoop`:

```
package com.alvinalexander.coinflip.v1

import CoinFlipUtils._
import scala.annotation.tailrec
import scala.util.Random
```

```
case class GameState(numFlips: Int, numCorrect: Int)

object CoinFlip extends App {

    val r = Random
    val s = GameState(0, 0)
    mainLoop(s, r)

    @tailrec
    def mainLoop(gameState: GameState, random: Random) {

        showPrompt()
        val userInput = getUserInput()

        // handle the result
        userInput match {
            case "H" | "T" => {
                val coinTossResult = tossCoin(random)
                val newNumFlips = gameState.numFlips + 1
                if (userInput == coinTossResult) {
                    val newNumCorrect = gameState.numCorrect + 1
             val newGameState = gameState.copy(numFlips = newNumFlips, numCorrect = newN
             printGameState(printableFlipResult(coinTossResult), newGameState)
                    mainLoop(newGameState, random)
                } else {
                 val newGameState = gameState.copy(numFlips = newNumFlips)
             printGameState(printableFlipResult(coinTossResult), newGameState)
                    mainLoop(newGameState, random)
                }
            }
            case _   => {
                printGameOver()
                printGameState(gameState)
                // return out of the recursion here
            }
```

```
        }
    }

}
```

There are a few ways to shorten and refactor that code, but it gives you an idea of what needs to be done for this game.

When the user's guess is correct

Note that when the user's guess matches the coin flip, I use this code:

```
val newNumCorrect = gameState.numCorrect + 1
val newGameState = gameState.copy(numFlips = newNumFlips, numCorrect = newNumCorrect)
printGameState(printableFlipResult(coinTossResult), newGameState)
mainLoop(newGameState, random)
```

The key here is that when the user's guess is correct I need to create a new `GameState` and pass that new instance into the next `mainLoop` call. I show that code in a long form, but I can remove the `newNumCorrect` temporary variable:

```
val newGameState = gameState.copy(
    numFlips = newNumFlips,
    numCorrect = gameState.numCorrect + 1
)
printGameState(printableFlipResult(coinTossResult), newGameState)
mainLoop(newGameState, random)
```

When the user's guess is incorrect

In the case where the user's guess is incorrect, I only need to update `numFlips` when creating a new `GameState` instance, so that block of code looks like this:

```
val newGameState = gameState.copy(numFlips = newNumFlips)
printGameState(printableFlipResult(coinTossResult), newGameState)
mainLoop(newGameState, random)
```

When the user wants to quit the game

In the case where the user enters anything other than `H` or `T`, I assume they want to quit the game, so I call these procedures:

```
printGameOver()
printGameState(gameState)
```

At this point I don't call `mainLoop` any more, so the recursion ends, all of the recursive calls unwind, and the game ends.

Summary

At the beginning of this lesson I noted that the goals for this lesson were:

- To write our first functional application
- Show a first example of how to handle "state" in an FP application

A few important parts about this lesson that you may not have seen before in traditional imperative code are:

- The use of an explicit `GameState` variable
- Using recursion as a way of looping
- The recursion let us define the `GameState` instance as an immutable `val` field

I'll come back to this example later in this book and show another way to handle the "main loop" without using recursion, but given what I've shown so far, recursion is the only way to write this code using only `val` fields.

Exercises

1. Modify the game so you can play a new game by pressing ‘n’
2. After adding the ability to play a new game, modify the program to keep a history of all previously-played games

See also

- The Procedure Syntax section of the Scala Style Guide
- A bug entry about deprecating the Procedure syntax
- How to prompt users for input from Scala shell scripts tutorial

42

A Quick Review of Case Classes

> *"The biggest advantage of case classes is that they support pattern matching."*
>
> — Programming in Scala

Goals

In this book I generally assume that you know the basics of the Scala programming language, but because `case` classes are so important to *functional programming in Scala* it's worth a quick review of what case classes are — the features they provide, and the benefits of those features.

Discussion

As opposed to a "regular" Scala `class`, a `case` class generates a lot of code for you, with the following benefits:

- An `apply` method is generated, so you don't need to use the `new` keyword to create a new instance of the class.
- *Accessor* methods are generated for each constructor parameter, because `case` class constructor parameters are public `val` fields by default.
- (You won't use `var` fields in this book, but if you did, *mutator* methods would also be generated for constructor parameters declared as `var`.)
- An `unapply` method is generated, which makes it easy to use `case` classes in `match` expressions. This is huge for Scala/FP.

- As you'll see in the next lesson, a `copy` method is generated. I never use this in Scala/OOP code, you'll use it all the time in Scala/FP.
- `equals` and `hashCode` methods are generated, which lets you compare objects and easily use them as keys in maps (and sets).
- A default `toString` method is generated, which is helpful for debugging.

A quick demo

To demonstrate how case classes work, here are a few examples that show each of these features and benefits in action.

No need for `new`

When you define a class as a `case` class, you don't have to use the `new` keyword to create a new instance:

```
scala> case class Person(name: String, relation: String)
defined class Person

// "new" not needed before Person
scala> val christina = Person("Christina", "niece")
christina: Person = Person(Christina,niece)
```

This is a nice convenience when writing Scala/OOP code, but it's a *terrific* feature when writing Scala/FP code, as you'll see throughout this book.

No mutator methods

Case class constructor parameters are `val` by default, so an *accessor* method is generated for each parameter, but mutator methods are not generated:

```
scala> christina.name
res0: String = Christina

// can't mutate the `name` field
scala> christina.name = "Fred"
<console>:10: error: reassignment to val
       christina.name = "Fred"
                      ^
```

`unapply` *method*

Because an `unapply` method is automatically created for a `case` class, it works well when you need to extract information in `match` expressions, as shown here:

```
scala> christina match { case Person(n, r) => println(n, r) }
(Christina,niece)
```

Conversely, if you try to use a regular Scala class in a `match` expression like this, you'll quickly see that it won't compile.

You'll see many more uses of case classes with `match` expressions in this book because *pattern macthing* is a BIG feature of Scala/FP.

> A class that defines an `unapply` method is called an *extractor*, and `unapply` methods enable match/case expressions. (I write more on this later in this book.)

`copy` *method*

A `case` class also has a built-in `copy` method that is extremely helpful when you need to clone an object and change one or more of the fields during the cloning process:

```
scala> case class BaseballTeam(name: String, lastWorldSeriesWin: Int)
defined class BaseballTeam

scala> val cubs1908 = BaseballTeam("Chicago Cubs", 1908)
cubs1908: BaseballTeam = BaseballTeam(Chicago Cubs,1908)

scala> val cubs2016 = cubs1908.copy(lastWorldSeriesWin = 2016)
cubs2016: BaseballTeam = BaseballTeam(Chicago Cubs,2016)
```

I refer to this process as "update as you copy," and this is such a big Scala/FP feature that I cover it in depth in the next lesson.

`equals` *and* `hashCode` *methods*

Case classes also have generated `equals` and `hashCode` methods, so instances can be compared:

```
scala> val hannah = Person("Hannah", "niece")
hannah: Person = Person(Hannah,niece)

scala> christina == hannah
res1: Boolean = false
```

These methods also let you easily use your objects in collections like sets and maps.

`toString` *methods*

Finally, `case` classes also have a good default `toString` method implementation, which at the very least is helpful when debugging code:

```
scala> christina
res0: Person = Person(Christina,niece)
```

Looking at the code generated by case classes

You can see the code that Scala case classes generate for you. To do this, first compile a simple `case` class, then disassemble the resulting *.class* files with `javap`.

For example, put this code in a file named *Person.scala*:

```
// note the `var` qualifiers
case class Person(var name: String, var age: Int)
```

Then compile it:

```
$ scalac Person.scala
```

`scalac` creates two JVM class files, *Person.class* and *Person$.class*. Disassemble *Person.class* with this command:

```
$ javap Person
```

With a few comments that I added, this command results in the following output, which is the public signature of the class:

```
Compiled from "Person.scala"
public class Person extends java.lang.Object implements scala.ScalaObject,scala.Proc
    public static final scala.Function1 tupled();
    public static final scala.Function1 curry();
    public static final scala.Function1 curried();
    public scala.collection.Iterator productIterator();
    public scala.collection.Iterator productElements();
    public java.lang.String name();           # getter
    public void name_$eq(java.lang.String);   # setter
    public int age();                         # getter
    public void age_$eq(int);                 # setter
    public Person copy(java.lang.String, int);
    public int copy$default$2();
    public java.lang.String copy$default$1();
    public int hashCode();
```

```
    public java.lang.String toString();
    public boolean equals(java.lang.Object);
    public java.lang.String productPrefix();
    public int productArity();
    public java.lang.Object productElement(int);
    public boolean canEqual(java.lang.Object);
    public Person(java.lang.String, int);
}
```

Next, disassemble Person$.class:

```
$ javap Person$

Compiled from "Person.scala"
public final class Person$ extends scala.runtime.AbstractFunction2 ▯
implements scala.ScalaObject,scala.Serializable{
    public static final Person$ MODULE$;
    public static {};
    public final java.lang.String toString();
    public scala.Option unapply(Person);
    public Person apply(java.lang.String, int);
    public java.lang.Object readResolve();
    public java.lang.Object apply(java.lang.Object, java.lang.Object);
}
```

As `javap` shows, Scala generates a *lot* of source code when you declare a class as a `case` class, including getter and setter methods, and the methods I mentioned: `copy`, `hashCode`, `equals`, `toString`, `unapply`, `apply`, and many more.

> As you see, case classes have even more methods, including `tupled`, `curry`, `curried`, etc. I discuss these other methods in this book as the need arises.

Case class compared to a "regular" class

As a point of comparison, if you remove the keyword `case` from that code — making it a "regular" Scala `class` — then compile it and disassemble it, you'll see that Scala only generates the following code:

```
public class Person extends java.lang.Object{
    public java.lang.String name();
    public void name_$eq(java.lang.String);
    public int age();
    public void age_$eq(int);
    public Person(java.lang.String, int);
}
```

As you can see, that's a BIG difference. The `case` class results in 22 more methods than the "regular" class. In Scala/OOP those extra fields are a nice convenience, but as you'll see in this book, these methods enable many essential FP features in Scala.

Summary

In this lesson I showed that the following methods are automatically created when you declare a class as a `case class`:

- `apply`
- `unapply`
- accessor methods are created for each constructor parameter
- `copy`
- `equals` and `hashCode`
- `toString`

These built-in methods make case classes easier to use in a functional programming style.

What's next

I thought it was worth this quick review of Scala `case` classes because the next thing we're going to do is dive into the `case` class `copy` method. Because you don't mutate objects in FP, you need to do something else to create updated instances of objects when things change, and the way you do this in Scala/FP is with the `copy` method.

See also

- Extractor objects in Scala
- Daniel Westheide has a good article on extractors

43

Update as You Copy, Don’t Mutate

> *“I’ve been imitated so well I’ve heard people copy my mistakes.”*
>
> Jimi Hendrix

Goals

In functional programming you don’t modify (mutate) existing objects, you create new objects with updated fields based on existing objects. For instance, last year my niece’s name was “Emily Means,” so I could have created a `Person` instance to represent her, like this:

```
val emily = Person("Emily", "Means")
```

Then she got married, and her last name became “Walls.” In an imperative programming language you would just change her last name, like this:

```
emily.setLastName("Walls")
```

But in FP you don’t do this, you don’t mutate existing objects. Instead, what you do is (a) you copy the existing object to a new object, and (b) during the copy process you update any fields you want to change by supplying their new values.

The way you do this in Scala/FP is with the `copy` method that comes with the Scala *case class*. This lesson shows a few examples of how to use `copy`, including how to use it with nested objects.

Source code

So you can follow along, the source code for this lesson is available at github.com/alvinj/FpUpdateAsYouCopy

Basic copy

When you're working with a simple object it's easy to use `copy`. Given a `case` class like this:

```
case class Person (firstName: String, lastName: String)
```

if you want to update a person's last name, you just "update as you copy," like this:

```
val emily1 = Person("Emily", "Means")
val emily2 = emily1.copy(lastName = "Walls")
```

As shown, in simple situations like this all you have to do to use `copy` is:

- Make sure your class is a `case` class.
- Create an initial object (`emily1`), as usual.
- When a field in that object needs to be updated, use `copy` to create a new object (`emily2`) from the original object, and specify the name of the field to be changed, along with its new value.

When you're updating one field, that's all you have to do.

> That's also all you have to do to update multiple fields, as I'll show shortly.

The original instance is unchanged

An important point to note about this is that the first instance remains unchanged. You can verify that by running a little `App` like this:

```
object CopyTest1 extends App {

    println("--- Before Copy ---")
    val emily1 = Person("Emily", "Means")
    println(s"emily1 = $emily1")

    // emily got married
    println("\n--- After Copy ---")
    val emily2 = emily1.copy(lastName = "Walls")
    println(s"emily1 = $emily1")
    println(s"emily2 = $emily2")

}
```

The output of `CopyTest1` looks as follows, showing that the original `emily1` instance is unchanged after the `copy`:

```
--- Before Copy ---
emily1 = Person(Emily,Means)

--- After Copy ---
emily1 = Person(Emily,Means)
emily2 = Person(Emily,Walls)
```

What happens in practice is that you discard the original object, so thinking about the old instance isn't typically an issue; I just want to mention it. (You'll see more examples of how this works as we go along.)

> In practice you also won't use intermediate variables with names like `emily1`, `emily2`, etc. We just need to do that now, until we learn a few more things.

Updating several attributes at once

It's also easy to update multiple fields at one time using `copy`. For instance, had `Person` been defined like this:

```
case class Person (
    firstName: String,
    lastName: String,
    age: Int
)
```

you could create an instance like this:

```
val emily1 = Person("Emily", "Means", 25)
```

and then create a new instance by updating several parameters at once, like this:

```
// emily is married, and a year older
val emily2 = emily1.copy(lastName = "Walls", age = 26)
```

That's all you have to do to update two or more fields in a simple `case` class.

Copying nested objects

As shown, using `copy` with simple `case` classes is straightforward. But when a `case` class contains other `case` classes, and those contain more `case` classes, things get more complicated and the required code gets more verbose.

For instance, let's say that you have a `case` class hierarchy like this:

```
case class BillingInfo(
    creditCards: Seq[CreditCard]
)

case class Name(
    firstName: String,
    mi: String,
    lastName: String
)

case class User(
```

```
    id: Int,
    name: Name,
    billingInfo: BillingInfo,
    phone: String,
    email: String
)

case class CreditCard(
    name: Name,
    number: String,
    month: Int,
    year: Int,
    cvv: String
)
```

Visually the relationship between these classes looks like Figure 43.1.

Notice a few things about this code:

- `User` has fields of type `Name` and `BillingInfo`
- `CreditCard` also has a field of the `Name` type

Despite a little complexity, creating an initial instance of `User` with this hierarchy is straightforward:

```
object NestedCopy1 extends App {

    val hannahsName = Name(
        firstName = "Hannah",
        mi = "C",
        lastName = "Jones"
    )

    // create a user
    val hannah1 = User(
        id = 1,
```

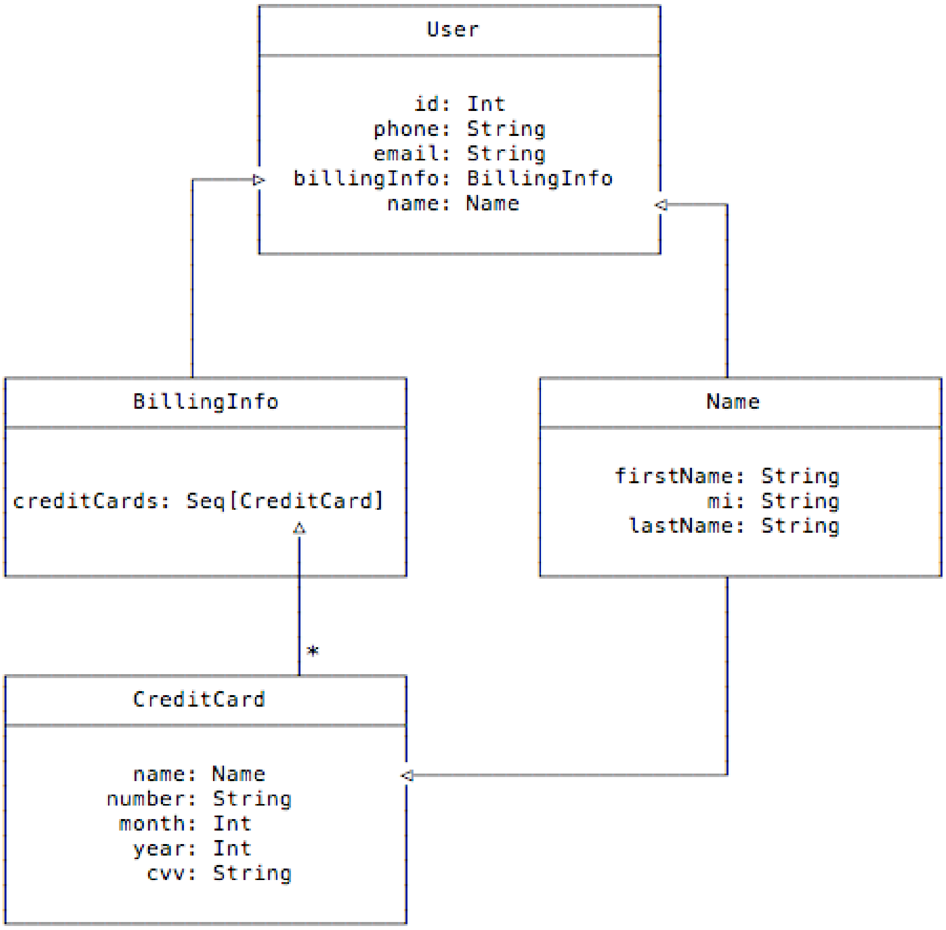


Figure 43.1: The visual relationship between the classes

```
        name = hannahsName,
        phone = "907-555-1212",
        email = "hannah@hannahjones.com",
        billingInfo = BillingInfo(
            creditCards = Seq(
                CreditCard(
                    name = hannahsName,
                    number = "1111111111111111",
                    month = 3,
                    year = 2020,
                    cvv = "123"
                )
            )
        )
    )

}
```

So far, so good. Now let's take a look at what you have to do when a few of the fields need to be updated.

Updating the phone number

First, let's suppose that Hannah moves. I kept the address out of the model to keep things relatively simple, but let's suppose that her phone number needs to be updated. Because the phone number is stored as a top-level field in `User`, this is a simple `copy` operation:

```
// hannah moved, update the phone number
val hannah2 = hannah1.copy(phone = "720-555-1212")
```

Updating the last name

Next, suppose that a little while later Hannah gets married and we need to update her last name. In this case you need to reach down into the `Name` instance of the `User` object and update the `lastName` field. I'll do this in a two-step process to keep it clear.

First, create a copy of the `name` field, changing `lastName` during the `copy` process:

```
// hannah got married, update her last name
val newName = hannah2.name.copy(lastName = "Smith")
```

If you print `newName` at this point, you'll see that it is "Hannah C Smith."

Now that you have this `newName` instance, the second step is to create a new "Hannah" instance with this new `Name`. You do that by (a) calling `copy` on the `hannah2` instance to make a new `hannah3` instance, and (b) within `copy` you bind the `name` field to `newName`:

```
val hannah3 = hannah2.copy(name = newName)
```

Updating the credit card

Suppose you also need to update the "Hannah" instance with new credit card information. To do this you follow the same pattern as before. First, you create a new `CreditCard` instance from the existing instance. Because the `creditCards` field inside the `billingInfo` instance is a `Seq`, you need to reference the first credit card instance while making the copy. That is, you reference `creditCards(0)`:

```
val oldCC = hannah3.billingInfo.creditCards(0)
val newCC = oldCC.copy(name = newName)
```

Because (a) `BillingInfo` takes a `Seq[CreditCard]`, and (b) there's only one credit card, I make a new `Seq[CreditCard]` like this:

```
val newCCs = Seq(newCC)
```

With this new `Seq[CreditCard]` I create a new "Hannah" instance by copying `hannah3` to `hannah4`, updating the `BillingInfo` during the `copy` process:

```
val hannah4 = hannah3.copy(billingInfo = BillingInfo(newCCs))
```

Put together, those lines of code look like this:

```
val oldCC = hannah3.billingInfo.creditCards(0)
val newCC = oldCC.copy(name = newName)
val newCCs = Seq(newCC)
val hannah4 = hannah3.copy(billingInfo = BillingInfo(newCCs))
```

You can shorten that code if you want, but I show the individual steps so it's easier to read.

These examples show how the "update as you copy" process works with nested objects in Scala/FP. (More on this after the attribution.)

Attribution

The examples I just showed are a simplification of the code and description found at these URLs:

- The "koffio-lenses" example on GitHub
- The KOFF.io "Lens in Scala" tutorial

Lenses

As you saw, the "update as you copy" technique gets more complicated when you deal with real-world, nested objects, and the deeper the nesting gets, the more complicated the problem becomes. But fear not: there are Scala/FP libraries that make this easier. The general idea of these libraries is known as a "lens" (or "lenses"), and they make copying nested objects much simpler. I cover lenses in a lesson later in this book.

Summary

Here's a summary of what I just covered:

- Because functional programmers don't mutate objects, when an object needs to be updated it's necessary to follow a pattern which I describe as "update as you copy".
- The way you do this in Scala is with the `copy` method, which comes with Scala `case` classes.
- As you can imagine, from here on out you're going to be using `case` classes more than you'll use the default Scala class. The `copy` method is just one reason for this, but it's a good reason. (You'll see even more reasons to use `case` classes as you go along.)

What's Next

As mentioned, I write about lenses later in the book, when we get to a point where we have to "update as you copy" complicated objects.

But for now the next thing we need to dig into is `for` comprehensions. Once I cover those, you'll be close to being able to write small, simple, functional applications with everything I've covered so far.

See Also

- The source code for this lesson is available at at this Github repository
- Alessandro Lacava has some notes about case classes, including a little about `copy`, currying, and arity
- The "koffio-lenses" example on GitHub
- The KOFF.io "Lens in Scala" tutorial

44

A Quick Review of `for` Expressions

This lesson is not included in the free preview.

45

How to Write a Class That Can Be Used in a `for` Expression

This lesson is not included in the free preview.

46

Creating a Sequence Class to be Used in a `for` Comprehension

This lesson is not included in the free preview.

47

Making Sequence Work in a Simple `for` Loop

This lesson is not included in the free preview.

48

How To Make Sequence Work as a Single Generator in a `for` Expression

This lesson is not included in the free preview.

49

Enabling Filtering in a `for` Expression

This lesson is not included in the free preview.

50

How to Enable the Use of Multiple Generators in a `for` Expression

This lesson is not included in the free preview.

51

A Summary of the `for` Expression Lessons

This lesson is not included in the free preview.

52

Pure Functions Tell No Lies

This lesson is not included in the free preview.

53

Functional Error Handling (Option, Try, or Either)

This lesson is not included in the free preview.

54

Embrace The Idioms!

This lesson is not included in the free preview.

55

What to Think When You See That Opening Curly Brace

This lesson is not included in the free preview.

56

A Quick Review of How `flatMap` Works

This lesson is not included in the free preview.

57

`Option` Naturally Leads to `flatMap`

This lesson is not included in the free preview.

58

`flatMap` Naturally Leads to `for`

This lesson is not included in the free preview.

59

`for` Expressions are Better Than `getOrElse`

This lesson is not included in the free preview.

60

Recap: `Option` -> `flatMap` -> `for`

This lesson is not included in the free preview.

61

A Note About Things That Can Be Mapped-Over

This lesson is not included in the free preview.

62

A Quick Review of Companion Objects and `apply`

This lesson is not included in the free preview.

63

Starting to Glue Functions Together

This lesson is not included in the free preview.

64

The “Bind” Concept

This lesson is not included in the free preview.

65

Getting Close to Using `bind` in `for` Expressions

This lesson is not included in the free preview.

66

Using a “Wrapper” Class in a `for` Expression

This lesson is not included in the free preview.

67

Making `Wrapper` More Generic

This lesson is not included in the free preview.

68

Changing “new Wrapper” to “Wrapper”

This lesson is not included in the free preview.

69

Using `bind` in a `for` Expression

This lesson is not included in the free preview.

70

How `Debuggable`, `f`, `g`, and `h` Work

This lesson is not included in the free preview.

71

A Generic Version of `Debuggable`

This lesson is not included in the free preview.

72

One Last `Debuggable`: Using `List` Instead of `String`

This lesson is not included in the free preview.

73

Key Points About Monads

This lesson is not included in the free preview.

74

Signpost: Where We're Going Next

This lesson is not included in the free preview.

75

Introduction: The IO Monad

This lesson is not included in the free preview.

76

How to Use an IO Monad

This lesson is not included in the free preview.

77

Assigning a `for` Expression to a Function

This lesson is not included in the free preview.

78

The `IO` Monad and a `for` Expression That Uses Recursion

This lesson is not included in the free preview.

79

Diving Deeper Into the IO Monad

This lesson is not included in the free preview.

80

I’ll Come Back to the IO Monad

This lesson is not included in the free preview.

81

Functional Composition

This lesson is not included in the free preview.

82

An Introduction to Handling State

This lesson is not included in the free preview.

83

Handling State Manually

This lesson is not included in the free preview.

84

Getting State Working in a `for` Expression

This lesson is not included in the free preview.

85

Handling My Golfing State with a State Monad

This lesson is not included in the free preview.

86

The `State` Monad Source Code

This lesson is not included in the free preview.

87

Signpost: Getting IO and State Working Together

This lesson is not included in the free preview.

88

Trying to Write a `for` Expression with IO and State

This lesson is not included in the free preview.

89

Seeing the Problem: Trying to Use State and IO Together

This lesson is not included in the free preview.

90

Solving the Problem with Monad Transformers

This lesson is not included in the free preview.

91

Beginning the Process of Understanding `StateT`

This lesson is not included in the free preview.

92

Getting Started: We’re Going to Need a Monad Trait

This lesson is not included in the free preview.

93

Now We Can Create StateT

This lesson is not included in the free preview.

94

Using StateT in a `for` Expression

This lesson is not included in the free preview.

95

Trying to Combine `IO` and `StateT` in a `for` Expression

This lesson is not included in the free preview.

96

Fixing the IO Functions With Monadic Lifting

This lesson is not included in the free preview.

97

A First IO/StateT `for` Expression

This lesson is not included in the free preview.

98

The Final IO/StateT `for` Expression

This lesson is not included in the free preview.

99

Summary of the `StateT` Lessons

This lesson is not included in the free preview.

100

Signpost: Modeling the world with Scala/FP

This lesson is not included in the free preview.

101

What is a Domain Model?

This lesson is not included in the free preview.

102

A Review of OOP Data Modeling

This lesson is not included in the free preview.

103

Modeling the "Data" Portion of the Pizza POS System with Scala/FP

This lesson is not included in the free preview.

104

First Attempts to Organize Pure Functions

This lesson is not included in the free preview.

105

Implementing FP Behavior with Modules

This lesson is not included in the free preview.

106

Implementing the Pizza POS System Using a Modular Approach

This lesson is not included in the free preview.

107

The "Functional Objects" Approach

This lesson is not included in the free preview.

108

Demonstrating the “Functional Objects” Approach

This lesson is not included in the free preview.

109

Summary of the Domain Modeling Approaches

This lesson is not included in the free preview.

110

ScalaCheck 1: Introduction

This lesson is not included in the free preview.

111

ScalaCheck 2: A More-Complicated Example

This lesson is not included in the free preview.

112

The Problem with the IO Monad

This lesson is not included in the free preview.

113

Signpost: Type Classes

This lesson is not included in the free preview.

114

Type Classes 101: Introduction

This lesson is not included in the free preview.

115

Type Classes 102: The Pizza Class

This lesson is not included in the free preview.

116

Type Classes 103: The Cats Library

This lesson is not included in the free preview.

117

Lenses, to Simplify “Update as You Copy”

This lesson is not included in the free preview.

118

Signpost: Concurrency

This lesson is not included in the free preview.

119

Concurrency and Mutability Don’t Mix

This lesson is not included in the free preview.

120

Scala Concurrency Tools

This lesson is not included in the free preview.

121

Akka Actors

This lesson is not included in the free preview.

122

Akka Actor Examples

This lesson is not included in the free preview.

123

Scala Futures

This lesson is not included in the free preview.

124

A Second Futures Example

This lesson is not included in the free preview.

125

Key Points About Futures

This lesson is not included in the free preview.

126

A Few Notes About Real World Functional Programming

This lesson is not included in the free preview.

127

Signpost: Wrapping Things Up

This lesson is not included in the free preview.

128

The Learning Path

This lesson is not included in the free preview.

129

Final Summary

This lesson is not included in the free preview.

130

Where To Go From Here

This lesson is not included in the free preview.

Appendices

This lesson is not included in the free preview.

A

Explaining Scala’s `val` Function Syntax

This lesson is not included in the free preview.

B

The Differences Between `val` and `def` When Creating Functions

This lesson is not included in the free preview.

C

A Review of Anonymous Functions

This lesson is not included in the free preview.

D

Recursion is Great, But …

This lesson is not included in the free preview.

E

`for` expression translation examples

This lesson is not included in the free preview.

F

On Using `def` vs `val` To Define Abstract Members in Traits

This lesson is not included in the free preview.

G

Algebraic Data Types

This lesson is not included in the free preview.

Index

??? syntax, 29

Akka, 51, 263
Alan Turing, 46
algebra, 38, 147
    definition, 149
    reason for “Going FP”, 148
algorithm, 332
Alonzo Church, 44, 46
always ask why, 19

best idea wins, 20
black holes and miracles, 181
book
    audience, 5
    concrete goals, 15
    goals, 9, 13
by-name parameters, 237
    background, 239
    with multiple parameter groups, 255
by-value parameters, 238

case class, 385
    copy method, 387
    javap, 389
    unapply method, 387
Cats, 15
Clojure
    concurrency, 69
Coin Flip game, 369
conservation of data, 179
control structures
    using, 259
    whilst, 254
    writing your own, 253
critical thinking, 22
curried functions
    creating, 279
Currying, 273
currying
    vs partially-applied functions, 284

data flow diagrams, 179
debugging is easier, 63
deterministic algorithms, 71
disclaimer, 17

ENIAC, 44
Erlang, 25, 50
Eta-expansion, 203
Expression-Oriented Programming, 161

FIP, function input parameter, 212
FP code
    concise, readable, 66
FP Terminology Barrier, 10
free variable, 118
function literal, 185
function signatures, 219
functional programming, 107
    benefits, 59, 76
    definition, 33, 36
    disadvantages, 79
    like Unix pipelines, 172
    math terminology, 81
    thought process, 176

functional programming as algebra, 147
functions
    passing around, 190
functions are variables, 183
Functor, 102
Future, 263

getStackTrace, 349

Haskell, 3, 51
hidden input, 118
Higher Order Functions, 211
higher-order function, 35
HOFs
    common control patterns, 215
    designing, 227

I/O wrapper code, 134
idempotent, 129
immutable variables
    benefits, 42
    parallel programming, 40
implicit execution context, 263
implicit variable
    multiple parameter groups, 260
implicit variables
    multiple in scope, 266
impure functions
    signs, 112
IO monad
    doesn't make a function pure, 86

javap, 389
javap -c, 22
Joe Armstrong, 25, 51
John Backus, 49
John McCarthy, 48
JVM
    stack, 338
    stack frame, 340

koan, 24

lambda, 43
lambda calculus, 43, 44
lambda means anonymous function, 44
lambda symbol, 44
LambdaConf, 15
Learning Cliff, 10
linked list
    cons cells, 296
Lisp, 48
list
    head, 296
    tail, 296
lists
    end with Nil, 311
    visualizing, 295
    ways to create, 300

map, 100
    writing a map function, 231
Martin Odersky, 53, 353
Mary Rose Cook, 37
memoization, 131
method
    convert to function manually, 204
methods
    using like functions, 199
monad
    first described, 56
multicore, 53
multiple parameter groups, 251

Niklaus Wirth, 53

OOP function signatures, 141

parallel programming, 68
parallel vs concurrent, 71
partially-applied function, 275
    example, 275
partially-applied functions
    and the JVM, 281
performance problems, 87
Pizza language, 53
procedures, 374
Prolog, 50
proofs, 63
property-based testing, 157
pseudocode, 372
pure function
    definition, 34, 41, 107
    mantra, 108
    signatures, 142
pure function game
    what can this function do?, 142
pure function signatures, 67
pure function signatures tell all, 146
pure functions, 60
    and I/O, 133
    benefits, 125
    examples, 110
pure functions and I/O, 84

question everything, 19

recursion, 81
    accumulator, 356
    case statements, 306
    conversation, 325
    how unwinding works, 313
    stack and stack frames, 342
    sum function, 303
    thought process, 329
    unwinding, 311, 348
    visualizing, 317
recursion is a by-product, 36
recursion, tail, 353
referential transparency, 129
reporter metaphor, 17
Richard Feynman, 20
rules for programming, 28

Scala/FP, 2
    function signatures, 73
Scala/FP disadvantages
    no standard FP library, 89
scalac
    compiler phases, 283
Scalaz, 15
Scheme, 49
side effects, 37
stack, 345
state, 365
    GameState, 366
statements vs expressions, 161
Swing
    invokeLater, 245

tail recursion, 353
Terminology Barrier, 99
testing is easier, 61
timer function, 240
transform as you copy, 155
type signatures, 221

Unit return type, 374
unit tests
    and purity, 120

Unix pipelines, 165
update as you copy, 82
    nested objects, 396
update as you copy, don’t mutate, 393
using control structure, 259

val function syntax, 186, 227

Walter Isaacson, 12, 56