Introducing Functional Programming with Underscore.js

Functional JavaScript

Michael Fogus
Forewords by Steve Vinoski & Jeremy Ashkenas

O'REILLY®

Functional JavaScript

Michael Fogus

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Functional JavaScript

by Michael Fogus

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Mary Treseler
Production Editor: Melanie Yarbrough
Copyeditor: Jasmine Kwityn
Proofreader: Jilly Gagnon
Indexer: Judith McConville
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

May 2013: First Edition

Revision History for the First Edition:

2013-05-24: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449360726* for release details.

ISBN: 978-1-449-36072-6

[LSI]

For Yuki

Table of Contents

Foreword by Jeremy Ashkenas

This is a terribly exciting book.

Despite its ignominious origins as a "Java-lite" scripting language, intended to be embedded inline in HTML documents to allow a minimum modicum of interactivity, JavaScript has always been one of the most essentially flexible languages for general purpose programming.

You can sketch, smudge, and draft bits of code in JavaScript, while pushing and twisting the language in the direction that best suits your particular style. The reason that this is more natural in JavaScript than in other, more rigid languages is due to the small set of strong core ideas that lie at the heart of JavaScript: Everything is an object (everything is a value) to an even greater extent than in famously object-oriented languages like Ruby and Java. Functions are objects, are values. An object may serve as prototype (default values) for any other object. There is only *one* kind of function, and depending on how you employ it, it can either serve as a pure function, a mutating procedure, or as a method on an object.

JavaScript enables, but does not enforce, many different programming styles. In the early days, we tended to bring our traditional expectations and "best" practices with us when we started to learn to write JavaScript. Naturally this led to much JavaScript resembling Java without the omnipresent types or even with the types still there, just living inside of annotation comments above each method. Gradually, experiments were made: folks started generating functions at runtime, working with immutable data structures, creating different patterns for object-orientation, discovering the magic of chaining APIs, or extending built-in prototypes with custom functionality.

One of my favorite recent developments is the enthusiastic embrace of functional programming ideas as appropriate tools for building rich JavaScript applications. As we move beyond form validation and DOM animation towards full-featured apps, where the JavaScript in your codebase might be getting up to any manner of hijinks in any particular problem space, functional ideas are similarly moving beyond the basic callback, and towards more interesting arenas, such as:

- Building out a large API by partially applying a core set of functions with arguments in different configurations.
- Using recursive functions to smooth the gap between actions that need to occur for a period of time, and events coming in rapid-fire off the event loop.
- Structuring a piece of complex business logic as a pipeline of mutation-free changes that can later be plugged-into and pulled apart.

You're reading the ideal book with which to explore this territory. In the following nine chapters (and two appendixes), your friendly tour guide and resident mad scientist, Michael Fogus, breaks down functional programming into its basic atoms, and builds it back up again into edifices of terrifying cleverness that will leave you wondering. It's rare that a programming book can take you by surprise, but this one will.

Enjoy.

—Jeremy Ashkenas

Foreword by Steve Vinoski

I remember when I first read Douglas Crockford's wonderful book *JavaScript: The Good Parts*. Not only did I learn from it, but the fact that Crockford required only 172 pages to steer readers away from JavaScript's problematic parts makes his work that much more impressive. Brevity is often at odds with educative exposition, but when an author achieves both as Crockford did, the reader is more likely to fully digest the author's recommendations and benefit from them.

In the pages that follow, you'll find that Michael Fogus has given us a book as excellent as Crockford's, perhaps more so. He's built on the sound advice of Crockford and other predecessors to take us on a deep dive into the world of functional JavaScript programming. I've often heard and read (and even written myself) that JavaScript is a functional programming language, but such assertions (including my own) have always seemed light on the pragmatic details that practicing programmers need. Even Crockford devoted only a single chapter to functions, focusing instead, like many authors, on JavaScript's object support. Here, merely saying that Fogus fills in those missing details would be a serious understatement.

Functional programming has been a part of the computing field from its inception, yet traditionally it has not enjoyed significant interest or growth among practicing software professionals. But thanks to continuing advances in computing hardware speed and capacity, coupled with our industry's increasing interest in creating software systems of ever-escalating levels of concurrency, distribution and scale, functional programming is rapidly growing in popularity. This growth is due to the observation that functonal programming appears to help developers reason about, build and maintain such systems. Curiosity about languages that support functional programming, like Scala, Clojure, Erlang and Haskell, is at an all-time high and still increasing, with no abatement in sight.

As you read through Michael's insightful investigations of JavaScript's functional programming capabilities, you'll be impressed with the significant depth and breadth of the information he provides. He keeps things simple at first, explaining how functions

and "data as abstraction" can avoid the desire to use JavaScript's powerful object prototype system to create yet another way of modeling classes. But as he explains and thoroughly reveals in subsequent chapters, the simple model of functional data transformation can yield sophisticated yet efficient building blocks and higher level abstractions. I predict you'll be amazed at just how far Fogus is able to take these innovative approaches as each chapter goes by.

Most software development efforts require pragmatism, though, and fortunately for us Fogus tackles this important requirement as well. Having beautiful, sophisticated and simple code is ultimately meaningless if it's not practical, and this is a large part of the reason functional programming stayed hidden in the shadows for so many years. Fogus addresses this issue by helping the reader explore and evaluate the computing costs associated with the functional programming approaches he champions here.

And of course books, just like software, are ultimately about communication. Like Crockford, Fogus writes in a manner that's both brief and informative, saying just enough to drive his ideas home without belaboring them. I can't overstate the importance of Michael's brevity and clarity, since without them we'd miss the incredible potential of the ideas and insights he's provided here. You'll find elegance not only in the approaches and code Fogus presents, but also in the way he presents them.

—Steve Vinoski

Preface

What Is Underscore?

Underscore.js (hereafter called Underscore) is a JavaScript library supporting functional programming. The Underscore website describes the library as such:

> Underscore is a utility-belt library for JavaScript that provides a lot of the functional programming support that you would expect in Prototype.js (or Ruby), but without extending any of the built-in JavaScript objects.

In case you didn't grow up watching the kitschy old Batman television show, the term "utility belt" means that it provides a set of useful tools that will help you solve many common problems.[1]

Getting Underscore

The Underscore website has the latest version of the library. You can download the source from the website and import it into the applicable project directories.

Using Underscore

Underscore can be added to your own projects in the same way you would add any other JavaScript library. However, there are a few points to make about how you interact with Underscore. First, by default Underscore defines a global object named _ that contains all of its functions. To call an Underscore function, you simply call it as a method on _, as shown in the following code:

1. Batman actually had more than just useful tools—he had tools for every conceivable circumstance, including those that might require a Bat Alphabet Soup Container or Bat Shark Repellant. Underscore doesn't quite match that level of applicability.

```
_.times(4, function() { console.log("Major") });

// (console) Major
// (console) Major
// (console) Major
// (console) Major
```

Simple, no?

One thing that might not be so simple is if you already defined a global _ variable. In this case, Underscore provides a `_.noConflict` function that will rebind your old _ and return a reference to Underscore itself. Therefore, using `_.noConflict` works as follows:

```
var underscore = _.noConflict();

underscore.times(4, function() { console.log("Major") });

// (console) Major
// (console) Major
// (console) Major
// (console) Major

_;
//=> Whatever you originally bound _ to
```

You'll see many of the details of Underscore throughout this book, but bear in mind that while I use Underscore extensively (and endorse it), this is not a book about Underscore per se.

The Source Code for Functional JavaScript

Many years ago, I wanted to write a library for JavaScript based on functional programming techniques. Like many programmers, I had obtained a working understanding of JavaScript through a mixture of experimentation, use, and the writing of Douglas Crockford. Although I went on to complete my functional library (which I named Doris), I rarely used it for even my own purposes.

After completing Doris, I went on to other ventures, including extensive work with (and on) the functional programming languages Scala and Clojure. Additionally, I spent a lot of time helping to write ClojureScript, especially its compiler that targets JavaScript. Based on these experiences, I gained a very good understanding of functional programming techniques. As a result, I decided to resurrect Doris and try it again, this time using techniques learned in the intervening years. The product of this effort was called Lemonad, which was developed in conjunction with the content of this book.

While many of the functions in this book are created for the purpose of illustration, I've expanded on the lessons in this book in my Lemonad library and the official underscore-contrib library.

Running the Code in This Book

The source code for *Functional JavaScript* is available on GitHub. Additionally, navigating to the book's website will allow you to use your browser's JavaScript console to explore the functions defined herein.

Notational Conventions

Throughout the course of this book (and in general when writing JavaScript) I observe various rules when writing functions, including the following:

- Avoid assigning variables more than once.
- Do not use `eval`.[2]
- Do not modify core objects like `Array` and `Function`.
- Favor functions over methods.
- If a function is defined at the start of a project, then it should work in subsequent stages as well.

Additionally, I use various conventions in the text of this book, including the following:

- Functions of zero parameters are used to denote that the arguments don't matter.
- In some examples, `...` is used to denote that the surrounding code segments are being ignored.
- Text like `inst#method` denotes a reference to an instance method.
- Text like `Object.method` denotes a reference to a type method.
- I tend to restrict `if/else` statements to a single line per branch, so I prefer to avoid using curly brackets to wrap the blocks. This saves precious vertical space.
- I like to use semicolons.

For the most part, the JavaScript code in this book is like the majority of JavaScript code that you'll see in the wild, except for the functional composition, which is the whole point of writing the book in the first place.

Whom Functional JavaScript Is Written For

This book started as an idea a few years ago, to write an introductory book on functional programming in the Scheme programming language. Although Scheme and JavaScript have some common features, they are very different in many important ways. However,

2. Like all powerful tools, JavaScript's `eval` and `Function` constructors can be used for harm as well as for good. I have nothing against them per se, but I rarely need them.

regardless of the language used, much of functional programming is transcendent. Therefore, I wrote this book to introduce functional programming in the context of what is and what is not possible with JavaScript.

I assume a base-level understanding of JavaScript. There are many amazing books on the topic and a bevy of online resources, so an introduction to the language is not provided herein. I also assume a working understanding of object-oriented programming, as commonly practiced in languages such as Java, Ruby, Python, and even JavaScript. While knowing object-oriented programming can help you to avoid my use of the occasional irrelevant phrase, an expert-level understanding of the subject is not required.

The ideal readers for *Functional JavaScript* are long-time JavaScript programmers hoping to learn about functional programming, or long-time functional programmers looking to learn JavaScript. For the latter case, it's advised that this book be supplemented with material focusing on JavaScript's…oddities. Of particular note is *JavaScript: The Good Parts* by Douglas Crockford (O'Reilly). Finally, this book is appropriate for anyone looking to learn more about functional programming, even those who have no intention of using JavaScript beyond the confines of these pages.

A Roadmap for Functional JavaScript

Here is an outline of the topics covered in *Functional JavaScript*:

Chapter 1, Introducing Functional JavaScript
: The book starts off by introducing some important topics, including functional programming and Underscore.js.

Chapter 2, First-Class Functions and Applicative Programming
: Chapter 2 defines first-class functions, shows how to use them, and describes some common applications. One particular technique using first-class functions—called applicative programming—is also described. The chapter concludes with a discussion of "data thinking," an important approach to software development central to functional programming.

Chapter 3, Variable Scope and Closures
: Chapter 3 is a transitional chapter that covers two topics of core importance to understanding functional programming in JavaScript. I start by covering variable scoping, including the flavors used within JavaScript: lexical scoping, dynamic scoping, and function scoping. The chapter concludes with a discussion of closures —how they operate, and how and why you might use them.

Chapter 4, Higher-Order Functions
: Building on the lessons of Chapters 2 and 3, this chapter describes an important type of first-class function: higher-order functions. Although "higher-order functions" sound complicated, this chapter shows that they are instead straightfoward.

Chapter 5, Function-Building Functions
: Moving on from the lessons of the previous chapters, Chapter 5 describes a way to "compose" functions from other functions. Composing functions is an important technique in functional programming, and this chapter will help guide you through the process.

Chapter 6, Recursion
: Chapter 6 is another transitional chapter in which I'll discuss recursion, a term that describes a function that calls itself either directly or indirectly. Because recursion is limited in JavaScript, it's not often used; however, there are ways around these limitations, and this chapter will guide you through a few.

Chapter 7, Purity, Immutability, and Policies for Change
: Chapter 7 deals with various ways to write functional code that doesn't change anything. Put simply, functional programming is facilitated when variables are not changed at all, and this chapter will guide you through just what that means.

Chapter 8, Flow-Based Programming
: Chapter 8 deals with viewing tasks, and even whole systems, as virtual "assembly lines" of functions that transform and move data.

Chapter 9, Programming Without Class
: The final chapter focuses on how functional programming allows you to structure applications in interesting ways that have nothing to do with class-based object-oriented programming.

Following these chapters, the book concludes with two appendixes of supplementary information: Appendix A, *Functional JavaScript in the Wild* and Appendix B, *Annotated Bibliography*.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

Using Code Examples

This book is here to help you get your job done. In general, if this book includes code examples, you may use the code in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Functional JavaScript* by Michael Fogus (O'Reilly). Copyright 2013 Michael Fogus, 978-1-449-36072-6."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/functional_js*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

It takes a village to write a book, and this book is no different. First, I would like to thank my good friend Rob Friesel for taking the time to provide feedback throughout the course of writing this book. Additionally, I would like to thank Jeremy Ashkenas for putting me in touch with O'Reilly and really making this book possible from the start. Plus he wrote the Underscore.js library—no small matter.

The following people have provided great conversation, direct feedback, or even inspiration from afar over the years, and I thank them all just for being awesome: Chris Houser, David Nolen, Stuart Halloway, Tim Ewald, Russ Olsen, Alan Kay, Peter Seibel, Sam Aaron, Brenton Ashworth, Craig Andera, Lynn Grogan, Matthew Flatt, Brian McKenna, Bodil Stokke, Oleg Kiselyov, Dave Herman, Mashaaricda Barmajada ee Mahmud, Patrick Logan, Alan Dipert, Alex Redington, Justin Gehtland, Carin Meier, Phil Bagwell, Steve Vinoski, Reginald Braithwaite, Daniel Friedman, Jamie Kite, William Byrd, Larry Albright, Michael Nygard, Sacha Chua, Daniel Spiewak, Christophe Grand, Sam Aaron, Meikel Brandmeyer, Dean Wampler, Clinton Dreisbach, Matthew Podwysocki, Steve Yegge, David Liebke, and Rich Hickey.

My soundtrack while writing *Functional JavaScript* was provided by Pantha du Prince, Black Ace, Brian Eno, Béla Bartók, Dieter Moebius, Sun Ra, Broadcast, Scientist, and John Coltrane.

Finally, nothing that I do would be possible without the support of the three loves of my life: Keita, Shota, and Yuki.

CHAPTER 1

Introducing Functional JavaScript

This chapter sets up the book in a number of important ways. In it, I will introduce Underscore and explain how you can start using it. Additionally, I will define the terms and goals of the rest of the book.

The Case for JavaScript

The question of why you might choose JavaScript is easily answered in a word: reach. In other words, aside from perhaps Java, there is no more popular programming language right now than JavaScript. Its ubiquity in the browser and its near-ubiquity in a vast sea of current and emerging technologies make it a nice—and sometimes the only—choice for portability.

With the reemergence of client-service and single-page application architectures, the use of JavaScript in discrete applications (i.e., single-page apps) attached to numerous network services is exploding. For example, Google Apps are all written in JavaScript, and are prime examples of the single-page application paradigm.

If you've come to JavaScript with a ready interest in functional programming, then the good news is that it supports functional techniques "right out of the box" (e.g., the function is a core element in JavaScript). For example, if you have any experience with JavaScript, then you might have seen code like the following:

```
[1, 2, 3].forEach(alert);
// alert box with "1" pops up
// alert box with "2" pops up
// alert box with "3" pops up
```

The `Array#forEach` method, added in the fifth edition of the ECMA-262 language standard, takes some function (in this case, `alert`) and passes each array element to the function one after the other. That is, JavaScript provides various methods and functions

that take other functions as arguments for some inner purpose. I'll talk more about this style of programming as the book progresses.

JavaScript is also built on a solid foundation of language primitives, which is amazing, but a double-edged sword (as I'll discuss soon). From functions to closures to prototypes to a fairly nice dynamic core, JavaScript provides a well-stocked set of tools.[1] In addition, JavaScript provides a very open and flexible execution model. As a small example, all JavaScript functions have an `apply` method that allows you to call the function with an array as if the array elements were the arguments to the function itself. Using `apply`, I can create a neat little function named `splat` that just takes a function and returns another function that takes an array and calls the original with `apply`, so that its elements serve as its arguments:

```
function splat(fun) {
  return function(array) {
    return fun.apply(null, array);
  };
}

var addArrayElements = splat(function(x, y) { return x + y });

addArrayElements([1, 2]);
//=> 3
```

This is your first taste of functional programming—a function that returns another function—but I'll get to the meat of that later. The point is that `apply` is only one of many ways that JavaScript is a hugely flexible programming language.

Another way that JavaScript proves its flexibility is that any function may be called with any number of arguments of any type, at any time. We can create a function `unsplat` that works opposite from `splat`, taking a function and returning another function that takes any number of arguments and calls the original with an array of the values given:

```
function unsplat(fun) {
  return function() {
    return fun.call(null, _.toArray(arguments));
  };
}

var joinElements = unsplat(function(array) { return array.join(' ') });

joinElements(1, 2);
//=> "1 2"

joinElements('-', '$', '/', '!', ':');
//=> "- $ / ! :"
```

1. And, as with all tools, you can get cut and/or smash your thumb if you're not careful.

Every JavaScript function can access a local value named `arguments` that is an array-like structure holding the values that the function was called with. Having access to `arguments` is surprisingly powerful, and is used to amazing effect in JavaScript in the wild. Additionally, the `call` method is similar to `apply` except that the former takes the arguments one by one rather than as an array, as expected by `apply`. The trifecta of `apply`, `call`, and `arguments` is only a small sample of the extreme flexibility provided by JavaScript.

With the emergent growth of JavaScript for creating applications of all sizes, you might expect stagnation in the language itself or its runtime support. However, even a casual investigation of the `ECMAScript.next` initiative shows that it's clear that JavaScript is an evolving (albeit slowly) language.[2] Likewise, JavaScript engines like V8 are constantly evolving and improving JavaScript speed and efficiency using both time-tested and novel techniques.

Some Limitations of JavaScript

The case against JavaScript—in light of its evolution, ubiquity, and reach—is quite thin. You can say much about the language quirks and robustness failings, but the fact is that JavaScript is here to stay, now and indefinitely. Regardless, it's worth acknowledging that JavaScript is a flawed language.[3] In fact, the most popular book on JavaScript, Douglas Crockford's *JavaScript: The Good Parts* (O'Reilly), spends more pages discussing the terrible parts than the good. The language has true oddities, and by and large is not particularly succinct in expression. However, changing the problems with JavaScript would likely "break the Web," a circumstance that's unacceptable to most. It's because of these problems that the number of languages targeting JavaScript as a compilation platform is growing; indeed, this is a very fertile niche.[4]

As a language supporting—and at times preferring—imperative programming techniques and a reliance on global scoping, JavaScript is unsafe by default. That is, building programs with a key focus on mutability is potentially confusing as programs grow. Likewise, the very language itself provides the building blocks of many high-level features found by default in other languages. For example, JavaScript itself, prior to trunk versions of ECMAScript 6, provides no module system, but facilitates their creation using raw objects. That JavaScript provides a loose collection of basic parts ensures a bevy of custom module implementations, each incompatible with the next.

2. A draft specification for `ES.next` is found at *http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts*.
3. The debate continues over just how deeply.
4. Some languages that target JavaScript include, but are not limited to, the following: ClojureScript, CoffeeScript, Roy, Elm, TypeScript, Dart, Flapjax, Java, and JavaScript itself!

Language oddities, unsafe features, and a sea of competing libraries: three legitimate reasons to think hard about the adoption of JavaScript. But there is a light at the end of the tunnel that's not just the light of an oncoming train. The light is that through discipline and an observance to certain conventions, JavaScript code can be not only safe, but also simple to understand and test, in addition to being proportionally scalable to the size of the code base. This book will lead you on the path to one such approach: functional programming.

Getting Started with Functional Programming

You may have heard of functional programming on your favorite news aggregation site, or maybe you've worked in a language supporting functional techniques. If you've written JavaScript (and in this book I assume that you have) then you indeed *have* used a language supporting functional programming. However, that being the case, you might not have used JavaScript in a functional way. This book outlines a functional style of programming that aims to simplify your own libraries and applications, and helps tame the wild beast of JavaScript complexity.

As a bare-bones introduction, functional programming can be described in a single sentence:

> Functional programming is the use of functions that transform values into units of abstraction, subsequently used to build software systems.

This is a simplification bordering on libel, but it's functional (ha!) for this early stage in the book. The library that I use as my medium of functional expression in JavaScript is Underscore, and for the most part, it adheres to this basic definition. However, this definition fails to explain the "why" of functional programming.

Why Functional Programming Matters

> The major evolution that is still going on for me is towards a more functional programming style, which involves unlearning a lot of old habits, and backing away from some OOP directions.
>
> —John Carmack

If you're familiar with object-oriented programming, then you may agree that its primary goal is to break a problem into parts, as shown in Figure 1-1 (Gamma 1995).

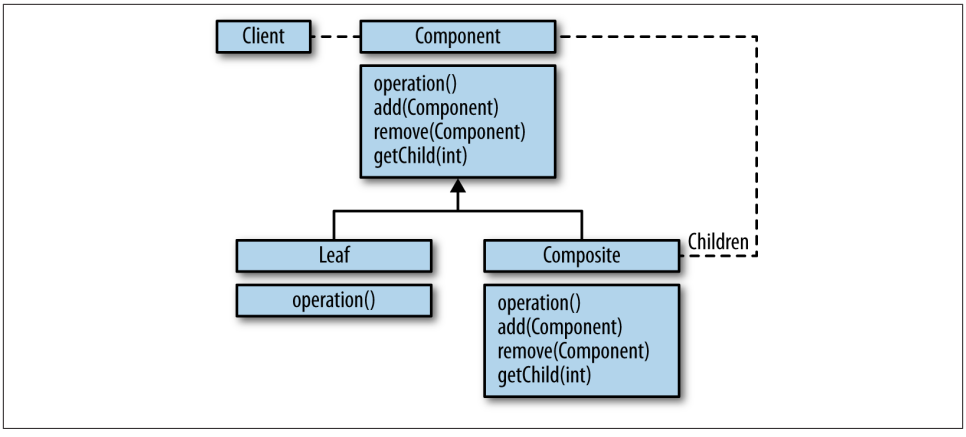

Figure 1-1. A problem broken into object-oriented parts

Likewise, these parts/objects can be aggregated and composed to form larger parts, as shown in Figure 1-2.

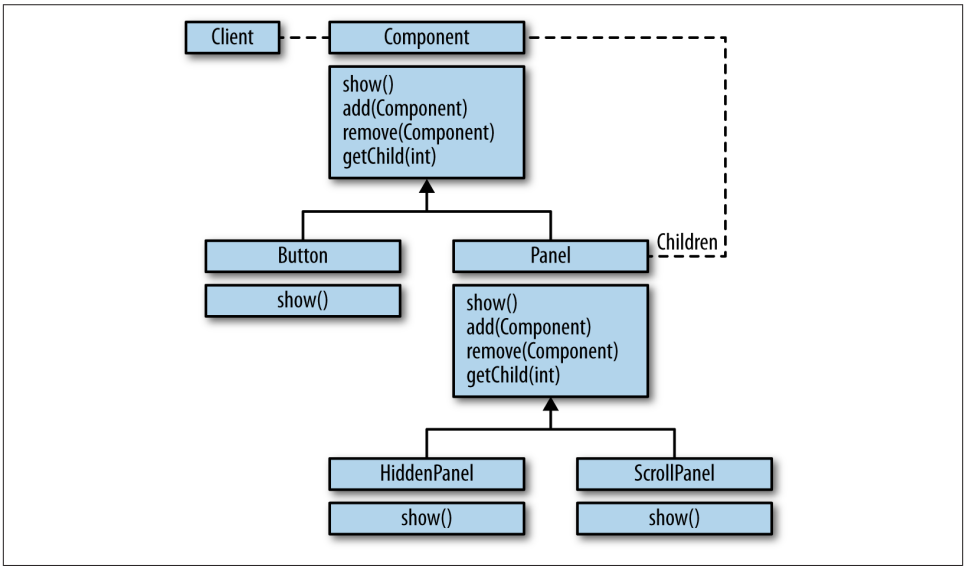

Figure 1-2. Objects are "composed" together to form bigger objects

Based on these parts and their aggregates, a system is then described in terms of the interactions and values of the parts, as shown in Figure 1-3.

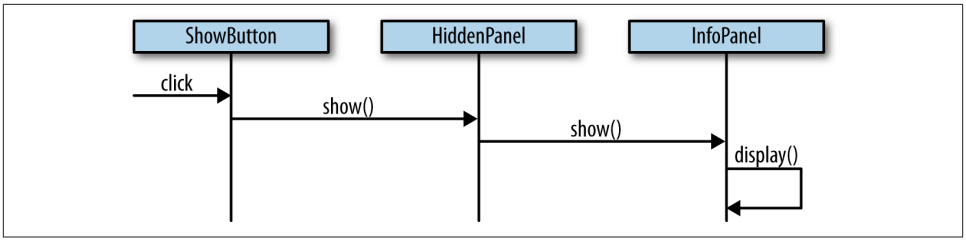

Figure 1-3. An object-oriented system and its interactions as a sequence diagram

This is a gross simplification of how object-oriented systems are formed, but I think that as a high-level description it works just fine.

By comparison, a strict functional programming approach to solving problems also breaks a problem into parts (namely, functions), as shown in Figure 1-4.

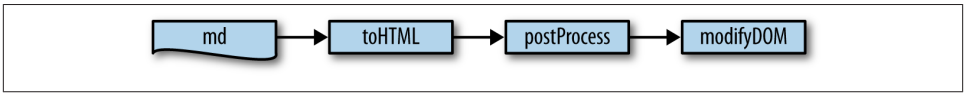

Figure 1-4. A problem broken into functional parts

Whereas the object-oriented approach tends to break problems into groupings of "nouns," or objects, a functional approach breaks the same problem into groupings of "verbs," or functions.[5] As with object-oriented programming, larger functions are formed by "gluing" or "composing" other functions together to build high-level behaviors, as shown in Figure 1-5.

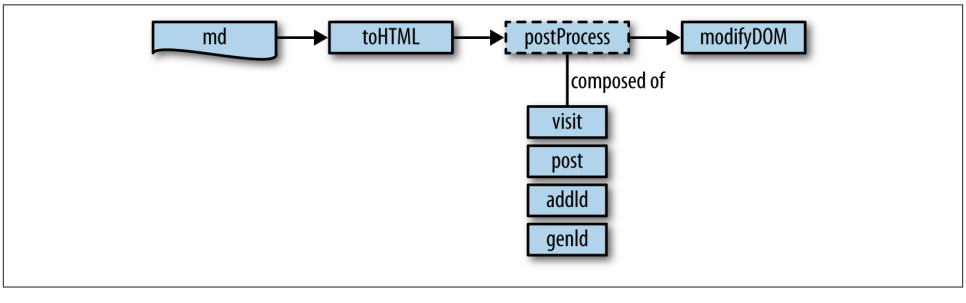

Figure 1-5. Functions are also composed together to form more behaviors

5. This is a simplistic way to view the composition of object-oriented versus functional systems, but bear with me as I develop a way to mix the two throughout the course of this book.

Finally, one way that the functional parts are formed into a system (as shown in Figure 1-6) is by taking a value and gradually "transforming" it—via one primitive or composed function—into another.

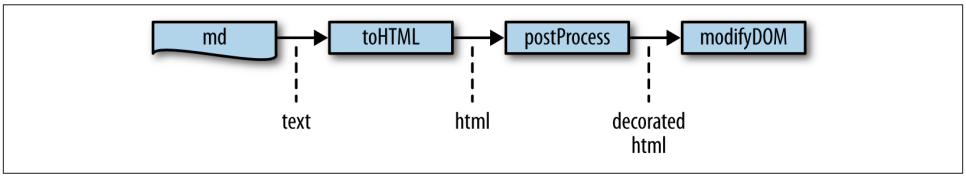

Figure 1-6. A functional system interacts via data transformation

In a system observing a strict object-oriented style, the interactions between objects cause internal change to each object, leading to an overall system state that is the amalgamation of many smaller, potentially subtle state changes. These interrelated state changes form a conceptual "web of change" that, at times, can be confusing to keep in your head. This confusion becomes a problem when the act of adding new objects and system features requires a working knowledge of the subtleties of potentially far-reaching state changes.

A functional system, on the other hand, strives to minimize observable state modification. Therefore, adding new features to a system built using functional principles is a matter of understanding how new functions can operate within the context of localized, nondestructive (i.e., original data is never changed) data transformations. However, I hesitate to create a false dichotomy and say that functional and object-oriented styles should stand in opposition. That JavaScript supports both models means that systems can and should be composed of both models. Finding the balance between functional and object-oriented styles is a tricky task that will be tackled much later in the book, when discussing mixins in Chapter 9. However, since this is a book about functional programming in JavaScript, the bulk of the discussion is focused on functional styles rather than object-oriented ones.

Having said that, a nice image of a system built along functional principles is an assembly-line device that takes raw materials in one end, and gradually builds a product that comes out the other end (Figure 1-7).

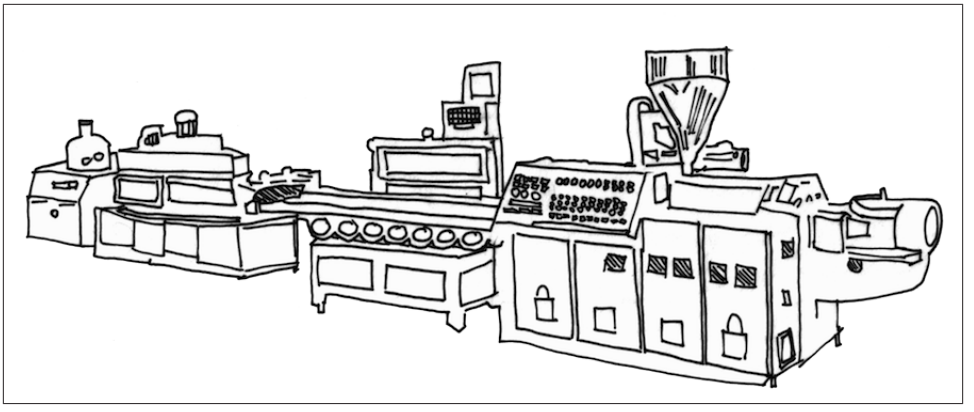

Figure 1-7. A functional program is a machine for transforming data

The assembly line analogy is, of course, not entirely perfect, because every machine I know consumes its raw materials to produce a product. By contrast, functional programming is what happens when you take a system built in an imperative way and shrink explicit state changes to the smallest possible footprint to make it more modular (Hughes 1984). Practical functional programming is not about eliminating state change, but instead about reducing the occurrences of mutation to the smallest area possible for any given system.

Functions as Units of Abstraction

One method of abstraction is that functions hide implementation details from view. In fact, functions are a beautiful unit of work allowing you to adhere to the long-practiced maxim in the UNIX community, set forth by Butler Lampson:

> Make it run, make it right, make it fast.

Likewise, functions-as-abstraction allow you to fulfill Kent Beck's similarly phrased mantra of test-driven development (TDD):

> Make it run, then make it right, then make it fast.

For example, in the case of reporting errors and warnings, you could write something like the following:

```
function parseAge(age) {
  if (!_.isString(age)) throw new Error("Expecting a string");
  var a;

  console.log("Attempting to parse an age");

  a = parseInt(age, 10);
```

```
  if (_.isNaN(a)) {
    console.log(["Could not parse age:", age].join(' '));
    a = 0;
  }

  return a;
}
```

This function, although not comprehensive for parsing age strings, is nicely illustrative. Use of `parseAge` is as follows:

```
parseAge("42");
// (console) Attempting to parse an age
//=> 42

parseAge(42);
// Error: Expecting a string

parseAge("frob");
// (console) Attempting to parse an age
// (console) Could not parse age: frob
//=> 0
```

The `parseAge` function works as written, but if you want to modify the way that errors, information, and warnings are presented, then changes need to be made to the appropriate lines therein, and anywhere else similar patterns are used. A better approach is to "abstract" the notion of errors, information, and warnings into functions:

```
function fail(thing) {
  throw new Error(thing);
}

function warn(thing) {
  console.log(["WARNING:", thing].join(' '));
}

function note(thing) {
  console.log(["NOTE:", thing].join(' '));
}
```

Using these functions, the `parseAge` function can be rewritten as follows:

```
function parseAge(age) {
  if (!_.isString(age)) fail("Expecting a string");
  var a;

  note("Attempting to parse an age");
  a = parseInt(age, 10);

  if (_.isNaN(a)) {
    warn(["Could not parse age:", age].join(' '));
    a = 0;
```

```
  }

  return a;
}
```

Here's the new behavior:

```
parseAge("frob");
// (console) NOTE: Attempting to parse an age
// (console) WARNING: Could not parse age: frob
//=> 0
```

It's not very different from the old behavior, except that now the idea of reporting errors, information, and warnings has been abstracted away. The reporting of errors, information, and warnings can thus be modified entirely:

```
function note() {}
function warn(str) {
  alert("That doesn't look like a valid age");
}

parseAge("frob");
// (alert box) That doesn't look like a valid age
//=> 0
```

Therefore, because the behavior is contained within a single function, the function can be replaced by new functions providing similar behavior or outright different behaviors altogether (Abelson and Sussman 1996).

Encapsulation and Hiding

Over the years, we've been taught that a cornerstone of object-oriented programming is *encapsulation*. The term encapsulation in reference to object-oriented programming refers to a way of packaging certain pieces of data with the very operations that manipulate them, as seen in Figure 1-8.

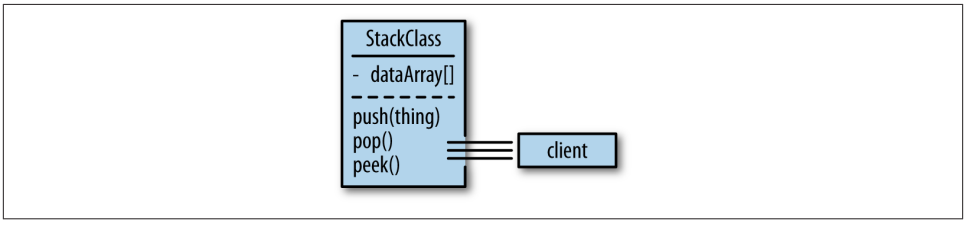

Figure 1-8. Most object-oriented languages use object boundaries to package data elements with the operations that work on them; a Stack class would therefore package an array of elements with the push, pop, and peek operations used to manipulate it

JavaScript provides an object system that does indeed allow you to encapsulate data with its manipulators. However, sometimes encapsulation is used to restrict the visibility of certain elements, and this act is known as *data hiding*. JavaScript's object system does not provide a way to hide data directly, so data is hidden using something called closures, as shown in Figure 1-9.

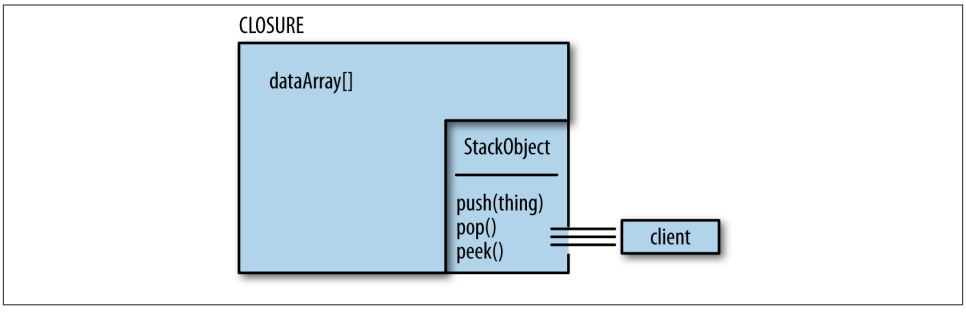

Figure 1-9. Using a closure to encapsulate data is a functional way to hide details from a client's view

Closures are not covered in any depth until Chapter 3, but for now you should keep in mind that closures are kinds of functions. By using functional techniques involving closures, you can achieve data hiding that is as effective as the same capability offered by most object-oriented languages, though I hesitate to say whether functional encapsulation or object-oriented encapsulation is better. Instead, while they are different in practice, they both provide similar ways of building certain kinds of abstraction. In fact, this book is not at all about encouraging you to throw away everything that you might have ever learned in favor of functional programming; instead, it's meant to explain functional programming on its own terms so that you can decide if it's right for your needs.

Functions as Units of Behavior

Hiding data and behavior (which has the side effect of providing a more agile change experience) is just one way that functions can be units of abstraction. Another is to provide an easy way to store and pass around discrete units of basic behavior. Take, for example, JavaScript's syntax to denote looking up a value in an array by index:

```
var letters = ['a', 'b', 'c'];

letters[1];
//=> 'b'
```

While array indexing is a core behavior of JavaScript, there is no way to grab hold of the behavior and use it as needed without placing it into a function. Therefore, a simple

example of a function that abstracts array indexing behavior could be called `nth`. The naive implementation of `nth` is as follows:

```
function naiveNth(a, index) {
  return a[index];
}
```

As you might suspect, `nth` operates along the happy path perfectly fine:

```
naiveNth(letters, 1);
//=> "b"
```

However, the function will fail if given something unexpected:

```
naiveNth({}, 1);
//=> undefined
```

Therefore, if I were to think about the abstraction surrounding a function `nth`, I might devise the following statement: *nth returns the element located at a valid index within a data type allowing indexed access*. A key part of this statement is the idea of an indexed data type. To determine if something is an indexed data type, I can create a function `isIndexed`, implemented as follows:

```
function isIndexed(data) {
  return _.isArray(data) || _.isString(data);
}
```

The function `isIndexed` is also a function providing an abstraction over checking if a piece of data is a string or an array. Building abstraction on abstraction leads to the following complete implementation of `nth`:

```
function nth(a, index) {
  if (!_.isNumber(index)) fail("Expected a number as the index");
  if (!isIndexed(a)) fail("Not supported on non-indexed type");
  if ((index < 0) || (index > a.length - 1))
    fail("Index value is out of bounds");

  return a[index];
}
```

The completed implementation of `nth` operates as follows:

```
nth(letters, 1);
//=> 'b'

nth("abc", 0);
//=> "a"

nth({}, 2);
// Error: Not supported on non-indexed type

nth(letters, 4000);
// Error: Index value is out of bounds
```

```
nth(letters, 'aaaaa');
// Error: Expected a number as the index
```

In the same way that I built the `nth` abstraction out of an `indexed` abstraction, I can likewise build a `second` abstraction:

```
function second(a) {
  return nth(a, 1);
}
```

The `second` function allows me to appropriate the correct behavior of `nth` for a different but related use case:

```
second(['a','b']);
//=> "b"

second("fogus");
//=> "o"

second({});
// Error: Not supported on non-indexed type
```

Another unit of basic behavior in JavaScript is the idea of a *comparator*. A comparator is a function that takes two values and returns `<1` if the first is less than the second, `>1` if it is greater, and `0` if they are equal. In fact, JavaScript itself can appear to use the very nature of numbers themselves to provide a default `sort` method:

```
[2, 3, -6, 0, -108, 42].sort();
//=> [-108, -6, 0, 2, 3, 42]
```

But a problem arises when you have a different mix of numbers:

```
[0, -1, -2].sort();
//=> [-1, -2, 0]

[2, 3, -1, -6, 0, -108, 42, 10].sort();
//=> [-1, -108, -6, 0, 10, 2, 3, 42]
```

The problem is that when given no arguments, the `Array#sort` method does a string comparison. However, every JavaScript programmer knows that `Array#sort` expects a comparator, and instead writes:

```
[2, 3, -1, -6, 0, -108, 42, 10].sort(function(x,y) {
  if (x < y) return -1;
  if (y < x) return  1;
  return 0;
});

//=> [-108, -6, -1, 0, 2, 3, 10, 42]
```

That seems better, but there is a way to make it more generic. After all, you might need to sort like this again in another part of the code, so perhaps it's better to pull out the anonymous function and give it a name:

```
function compareLessThanOrEqual(x, y) {
  if (x < y) return -1;
  if (y < x) return  1;
  return 0;
}

[2, 3, -1, -6, 0, -108, 42, 10].sort(compareLessThanOrEqual);
//=> [-108, -6, -1, 0, 2, 3, 10, 42]
```

But the problem with the `compareLessThanOrEqual` function is that it is coupled to the idea of "comparatorness" and cannot easily stand on its own as a generic comparison operation:

```
if (compareLessThanOrEqual(1,1))
  console.log("less or equal");

// nothing prints
```

To achieve the desired effect, I would need to *know* about `compareLessThanOrEqual`'s comparator nature:

```
if (_.contains([0, -1], compareLessThanOrEqual(1,1)))
  console.log("less or equal");

// less or equal
```

But this is less than satisfying, especially when there is a possibility for some developer to come along in the future and change the return value of `compareLessThanOrEqual` to `-42` for negative comparisons. A better way to write `compareLessThanOrEqual` might be as follows:

```
function lessOrEqual(x, y) {
  return x <= y;
}
```

Functions that always return a Boolean value (i.e., `true` or `false` only), are called *predicates*. So, instead of an elaborate comparator construction, `lessOrEqual` is simply a "skin" over the built-in `<=` operator:

```
[2, 3, -1, -6, 0, -108, 42, 10].sort(lessOrEqual);
//=> [100, 10, 1, 0, -1, -1, -2]
```

At this point, you might be inclined to change careers. However, upon further reflection, the result makes sense. If `sort` expects a comparator, and `lessThan` only returns `true` or `false`, then you need to somehow get from the world of the latter to that of the former without duplicating a bunch of `if/then/else` boilerplate. The solution lies in creating a function, `comparator`, that takes a predicate and converts its result to the `-1/0/1` result expected of `comparator` functions:

```
function comparator(pred) {
  return function(x, y) {
    if (truthy(pred(x, y)))
```

```
      return -1;
    else if (truthy(pred(y, x)))
      return 1;
    else
      return 0;
  };
};
```

Now, the `comparator` function can be used to return a new function that "maps" the results of the predicate `lessOrEqual` (i.e., `true` or `false`) onto the results expected of comparators (i.e., `-1`, `0`, or `1`), as shown in Figure 1-10.

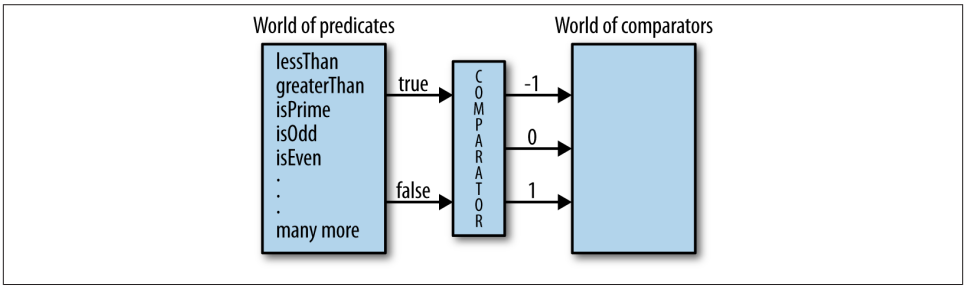

Figure 1-10. Bridging the gap between two "worlds" using the comparator function

In functional programming, you'll almost always see functions interacting in a way that allows one type of data to be brought into the world of another type of data. Observe `comparator` in action:

```
[100, 1, 0, 10, -1, -2, -1].sort(comparator(lessOrEqual));
//=> [-2, -1, -1, 0, 1, 10, 100]
```

The function `comparator` will work to map any function that returns "truthy" or "falsey" values onto the notion of "comparatorness." This topic is covered in much greater depth in Chapter 4, but it's worth noting now that `comparator` is a *higher-order function* (because it takes a function and returns a new function). Keep in mind that not every predicate makes sense for use with the `comparator` function, however. For example, what does it mean to use the `_.isEqual` function as the basis for a `comparator`? Try it out and see what happens.

Throughout this book, I will talk about the ways that functional techniques provide and facilitate the creation of abstractions, and as I'll discuss next, there is a beautiful synergy between functions-as-abstraction and data.

Data as Abstraction

JavaScript's object prototype model is a rich and foundational data scheme. On its own, the prototype model provides a level of flexibility not found in many other mainstream

programming languages. However, many JavaScript programmers, as is their wont, immediately attempt to build a class-based object system using the prototype or closure features (or both).[6] Although a class system has its strong points, very often the data needs of a JavaScript application are much simpler than is served by classes.[7]

Instead, using JavaScript bare data primitives, objects, and arrays, much of the data modeling tasks that are currently served by classes are subsumed. Historically, functional programming has centered around building functions that work to achieve higher-level behaviors and work on very simple data constructs.[8] In the case of this book (and Underscore itself), the focus is indeed on processing arrays and objects. The flexibility in those two simple data types is astounding, and it's unfortunate that they are often overlooked in favor of yet another class-based system.

Imagine that you're tasked with writing a JavaScript application that deals with comma-separated value (CSV) files, which are a standard way to represent data tables. For example, suppose you have a CSV file that looks as follows:

```
name,   age, hair
Merble, 35,  red
Bob,    64,  blonde
```

It should be clear that this data represents a table with three columns (name, age, and hair) and three rows (the first being the header row, and the rest being the data rows). A small function to parse this very constrained CSV representation stored in a string is implemented as follows:

```
function lameCSV(str) {
  return _.reduce(str.split("\n"), function(table, row) {
    table.push(_.map(row.split(","), function(c) { return c.trim()}));
    return table;
  }, []);
};
```

You'll notice that the function `lameCSV` processes the rows one by one, splitting at `\n` and then stripping whitespace for each cell therein.[9] The whole data table is an array of sub-arrays, each containing strings. From the conceptual view shown in Table 1-1, nested arrays can be viewed as a table.

6. The `ECMAScript.next` initiative is discussing the possibility of language support for classes. However, for various reasons outside the scope of this book, the feature is highly controversial. As a result, it's unclear when and if classes will make it into JavaScript core.

7. One strong argument for a class-based object system is the historical use in implementing user interfaces.

8. Very often you will see a focus on list data structures in functional literature. In the case of JavaScript, the array is a nice substitute.

9. The function `lameCSV` is meant for illustrative purposes and is in no way meant as a fully featured CSV parser.

Table 1-1. Simply nested arrays are one way to abstract a data table

name	age	hair
Merble	35	red
Bob	64	blonde

Using `lameCSV` to parse the data stored in a string works as follows:

```
var peopleTable = lameCSV("name,age,hair\nMerble,35,red\nBob,64,blonde");

peopleTable;
//=> [["name",   "age",   "hair"],
//    ["Merble", "35",   "red"],
//    ["Bob",    "64",   "blonde"]]
```

Using selective spacing highlights the table nature of the returned array. In functional programming, functions like `lameCSV` and the previously defined `comparator` are key in translating one data type into another. Figure 1-11 illustrates how data transformations in general can be viewed as getting from one "world" into another.

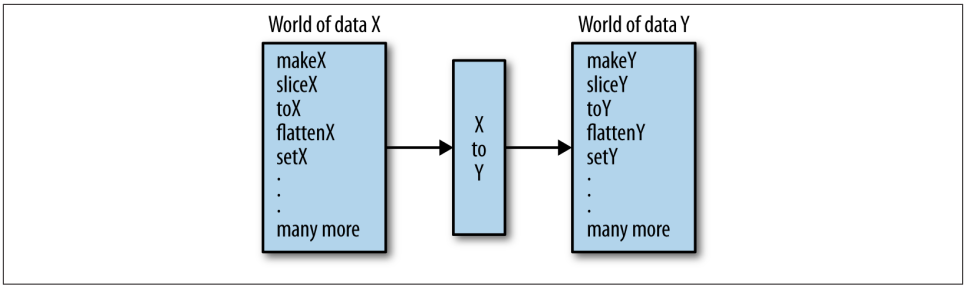

Figure 1-11. Functions can bridge the gap between two "worlds"

There are better ways to represent a table of such data, but this nested array serves us well for now. Indeed, there is little motivation to build a complex class hierarchy representing either the table itself, the rows, people, or whatever. Instead, keeping the data representation minimal allows me to use existing array fields and array processing functions and methods out of the box:

```
_.rest(peopleTable).sort();

//=> [["Bob",    "64",   "blonde"],
//    ["Merble", "35",   "red"]]
```

Likewise, since I know the form of the original data, I can create appropriately named selector functions to access the data in a more descriptive way:

```
function selectNames(table) {
  return _.rest(_.map(table, _.first));
}
```