CHAPTER 2

First-Class Functions and Applicative Programming

The basics of functional programming, which treats functions as first-class elements of a language, are covered in this chapter. I'll provide a basis in the three common functions: map, reduce, and filter. Since programmers will likely be familiar with these functions, using them as a starting point should provide a nice foundation for the rest of the book.

Functions as First-Class Things

Some programmers familiar with JavaScript, myself included, consider it to be a functional language. Of course, to say such a thing implies that others disagree with that assessment. The reason for this disagreement stems from the fact that functional programming often has a relative definition, differing in minor and major ways from one practitioner or theorist to another.[1]

This is a sad state of affairs, indeed. Thankfully, however, almost every single relative definition of functional programming seems to agree on one point: a functional programming language is one facilitating the use and creation of first-class functions.

Typically, you will see this point accompanied by other definitional qualifications including but not limited to static typing, pattern matching, immutability, purity, and so on. However, while these other points describe certain implementations of functional programming languages, they fail in broad applicability. If I boil down the definition to its essence, consisting of the terms "facilitating" and "first-class functions," then it covers a broad range of languages from Haskell to JavaScript—the latter being quite important

1. Aside from the fact that programmers rarely agree on even the most common terms.

to this book. Thankfully, this also allows first-class functions to be defined in a paragraph.[2]

The term "first-class" means that something is just a value. A first-class function is one that can go anywhere that any other value can go—there are few to no restrictions. A number in JavaScript is surely a first-class thing, and therefore a first-class function has a similar nature:

- A number can be stored in a variable and so can a function:

```
var fortytwo = function() { return 42 };
```

- A number can be stored in an array slot and so can a function:

```
var fortytwos = [42, function() { return 42 }];
```

- A number can be stored in an object field and so can a function:

```
var fortytwos = {number: 42, fun: function() { return 42 }};
```

- A number can be created as needed and so can a function:

```
42 + (function() { return 42 })();
//=> 84
```

- A number can be passed to a function and so can a function:

```
function weirdAdd(n, f) { return n + f() }

weirdAdd(42, function() { return 42 });
//=> 84
```

- A number can be returned from a function and so can a function:

```
return 42;

return function() { return 42 };
```

The last two points define by example what we would call a "higher-order" function; put directly, a higher-order function can do one or both of the following:

- Take a function as an argument
- Return a function as a result

2. Haskell programs dealing with matters of I/O are often highly imperative in nature, but you would be hard-pressed to find someone claiming that Haskell was not functional.

In Chapter 1, `comparator` was used as an example of a higher-order function, but here is yet another example:

```
_.each(['whiskey', 'tango', 'foxtrot'], function(word) {
  console.log(word.charAt(0).toUpperCase() + word.substr(1));
});

// (console) Whiskey
// (console) Tango
// (console) Foxtrot
```

Underscore's `_.each` function takes a collection (object or array) and loops over its elements, calling the function given as the second argument for *each* element.

I'll dive deeper into higher-order functions in Chapter 4. For now, I'll take a couple of pages to talk about JavaScript itself, because as you may already know, while it supports a functional style, it also supports a number of other programming paradigms.

JavaScript's Multiple Paradigms

Of course JavaScript is not strictly a functional programming language, but instead facilitates the use of other paradigms as well:

Imperative programming
: Programming based around describing actions in detail

Prototype-based object-oriented programming
: Programming based around prototypical objects and instances of them

Metaprogramming
: Programming manipulating the basis of JavaScript's execution model

Including only imperative, object-oriented, and metaprogramming restricts us to only those paradigms directly supported by the built-in language constructs. You could further support other paradigms, like class orientation and evented programming, using the language itself as an implementation medium, but this book does not deal with those topics in depth. Before I get into the definition and details of JavaScript's support for first-class functions, let me take a brief moment to elucidate how the other three models differ from functional programming. I'll dig deeper into each topic throughout this book, so for now a paragraph or two on each should suffice in transitioning you into the functional programming discussion.

Imperative programming

An imperative programming style is categorized by its exquisite (and often infuriating) attention to the details of algorithm implementation. Further, imperative programs are often built around the direct manipulation and inspection of program state. For example, imagine that you'd like to write a program to build a lyric sheet for the song "99

Bottles of Beer." The most direct way to describe the requirements of this program are as such:

- Start at 99
- Sing the following for each number down to 1:
  - — X bottles of beer on the wall
  - — X bottles of beer
  - — Take one down, pass it around
  - — X-1 bottles of beer on the wall
- Subtract one from the last number and start over with the new value
- When you finally get to the number 1, sing the following last line instead:
  - — No more bottles of beer on the wall

As it turns out, this specification has a fairly straightforward imperative implementation in JavaScript, as shown here:

```
var lyrics = [];

for (var bottles = 99; bottles > 0; bottles--) {
  lyrics.push(bottles + " bottles of beer on the wall");
  lyrics.push(bottles + " bottles of beer");
  lyrics.push("Take one down, pass it around");

  if (bottles > 1) {
    lyrics.push((bottles - 1) + " bottles of beer on the wall.");
  }
  else {
    lyrics.push("No more bottles of beer on the wall!");
  }
}
```

This imperative version, while somewhat contrived, is emblematic of an imperative programming style. That is, the implementation describes a "99 Bottles of Beer" program and exactly a "99 Bottles of Beer" program. Because imperative code operates at such a precise level of detail, they are often one-shot implementations or at best, difficult to reuse. Further, imperative languages are often restricted to a level of detail that is good for their compilers rather than for their programmers (Sokolowski 1991).

By comparison, a more functional approach to this same problem might look as follows:

```
function lyricSegment(n) {
  return _.chain([])
    .push(n + " bottles of beer on the wall")
    .push(n + " bottles of beer")
    .push("Take one down, pass it around")
    .tap(function(lyrics) {
```

```
            if (n > 1)
              lyrics.push((n - 1) + " bottles of beer on the wall.");
            else
              lyrics.push("No more bottles of beer on the wall!");
          })
      .value();
}
```

The `lyricSegment` function does very little on its own—in fact, it only generates the lyrics for a single verse of the song for a given number:

```
lyricSegment(9);

//=> ["9 bottles of beer on the wall",
//    "9 bottles of beer",
//    "Take one down, pass it around",
//    "8 bottles of beer on the wall."]
```

Functional programming is about pulling programs apart and reassembling them from the same parts, abstracted behind function boundaries. Thinking in this way, you can imagine that the `lyricSegment` function is the part of the "99 Bottles" program that abstracts lyric generation. Therefore, the part of the program that abstracts the assembly of the verse segments into a song is as follows:

```
function song(start, end, lyricGen) {
  return _.reduce(_.range(start,end,-1),
    function(acc,n) {
      return acc.concat(lyricGen(n));
    }, []);
}
```

And using it is as simple as:

```
song(99, 0, lyricSegment);

//=> ["99 bottles of beer on the wall",
//    ...
//    "No more bottles of beer on the wall!"]
```

Abstracting in this way allows you to separate out domain logic (i.e., the generation of a lyrical verse) from the generic verse assembly machinery. If you were so inclined, you could pass different functions like `germanLyricSegment` or `agreementLyricSegment` into `song` to generate a different lyric sheet altogether. Throughout this book, I'll use this technique, and explain it in greater depth along the way.

Prototype-based object-oriented programming

JavaScript is very similar to Java or C# in that its constructor functions are classes (at least at the level of implementation details), but the method of use is at a lower level. Whereas every instance in a Java program is generated from a class serving as its template, JavaScript instances use existing objects to serve as prototypes for specialized

instances.[3] Object specialization, together with a built-in dispatch logic that routes calls down what's called a *prototype chain,* is far more low-level than class-oriented programming, but is extremely elegant and powerful. I will talk about exploiting JavaScript's prototype chain later in Chapter 9.

For now, how this relates to functional programming is that functions can also exist as values of object fields, and Underscore itself is the perfect illustration:

```
_.each;

//=> function (array, n, guard) {
//    ...
//   }
```

This is great and beautiful, right? Well…not exactly. You see, because JavaScript is oriented around objects, it must have a semantics for self-references. As it turns out, its self-reference semantics conflict with the notion of functional programming. Observe the following:

```
var a = {name: "a", fun: function () { return this; }};

a.fun();
//=> {name: "a", fun: ...};
```

You'll notice that the self-reference `this` returned from the embedded `fun` function returns the object `a` itself. This is probably what you would expect. However, observe the following:

```
var bFunc = function () { return this };
var b = {name: "b", fun: bFunc};

b.fun();
//=> some global object, probably Window
```

Well, this is surprising. You see, when a function is created outside of the context of an object instance, its `this` reference points to the global object. Therefore, when I later bound `bFunc` to the field `b.fun`, its reference was never updated to `b` itself. In most programming languages offering both functional and object-oriented styles, a trade-off is made in the way that self-reference is handled. JavaScript has its approach while Python has a different approach and Scala has a different approach still. Throughout this book, you'll notice a fundamental tension between an object-oriented style and a functional style, but Underscore provides some tools to relieve, if not eliminate, this tension. This will be covered in greater depth later, but for now keep in mind that when I use the word "function" I mean a function that exists on its own and when I use "method" I mean a function created in the context of an object.

3. There is an existential crisis lurking in the question, "who created the first object?"

Metaprogramming

Related to prototype-based object-oriented programming are the facilities provided by JavaScript supporting metaprogramming. Many programming languages support metaprogramming, but rarely do they provide the level of power offered by JavaScript. A good definition of metaprogramming goes something like this: programming occurs when you write code to do something and metaprogramming occurs when you write code that changes the way that something is interpreted. Let's take a look at an example of metaprogramming so that you can better understand.

In the case of JavaScript, the dynamic nature of the `this` reference can be exploited to perform a bit of metaprogramming. For example, observe the following constructor function:

```
function Point2D(x, y) {
  this._x = x;
  this._y = y;
}
```

When used with `new`, the `Point2D` function gives a new object instance with the proper fields set as you might expect:

```
new Point2D(0, 1);

//=> {_x: 0, _y: 1}
```

However, the `Function.call` method can be used to metaprogram a derivation of the `Point2D` constructor's behavior for a new `Point3D` type:

```
function Point3D(x, y, z) {
  Point2D.call(this, x, y);
  this._z = z;
}
```

And creating a new instance works like a champ:

```
new Point3D(10, -1, 100);

//=> {_x: 10, _y: -1, _z: 100}
```

Nowhere did `Point3D` explicitly set the values for `this._x` and `this._y`, but by dynamically binding the `this` reference in a call to `Point2D` it became possible to change the target of its property creation code.

I will not go too deeply into JavaScript metaprogramming in this book because it's orthogonal to functional programming, but I'll take advantage of it occasionally throughout this book.[4]

4. If you're interested in a great book about JavaScript metaprogramming, then petition O'Reilly to have me write it. ☺

Applicative Programming

So far in this book I've shown only one aspect of functional programming that deals with a narrow band of the capabilities of functions—namely, applicative programming. Applicative programming is defined as the calling by function B of a function A, supplied as an argument to function B originally. I will not use the term "applicative" very often in this book because variations of that word can appear in different contexts with different meanings, but it's good to know should you see it in the future. That said, the three canonical examples of applicative functions are `map`, `reduce`, and `filter`. Observe how they operate:

```
var nums = [1,2,3,4,5];

function doubleAll(array) {
  return _.map(array, function(n) { return n*2 });
}

doubleAll(nums);
//=> [2, 4, 6, 8, 10]

function average(array) {
  var sum = _.reduce(array, function(a, b) { return a+b });
  return sum / _.size(array);
}

average(nums);
//=> 3

/* grab only even numbers in nums */
function onlyEven(array) {
  return _.filter(array, function(n) {
    return (n%2) === 0;
  });
}

onlyEven(nums);
//=> [2, 4]
```

You can imagine that somewhere inside of `map`, `reduce`, and `filter` a call to the function that's passed in occurs, and indeed that is the case. In fact, the semantics of these functions can be defined in terms of that very relationship:

- `_.map` calls a function on every value in a collection in turn, returning a collection of the results
- `_.reduce` collects a composite value from the incremental results of a function supplied with an accumulation value and each value in a collection

- `_.filter` calls a predicate function (one returning a true or false value) and grabs each value where said predicate returned `true`, returning them in a new collection

The functions `map`, `reduce`, and `filter` are as simple and as emblematic of applicative functional programming as you can get, but Underscore provides numerous others for your use. Before I get into those, let me take a moment to cover the idea of collection-centric programming, which is often coupled with functional programming itself.

Collection-Centric Programming

Functional programming is extremely useful for tasks requiring that some operation happen on many items in a collection. Certainly an array of numbers `[1,2,3,4,5]` is a collection of numbers, but we can also envision that an object `{a: 1, b: 2}` is a collection of key/value pairs. Take the simple case of `_.map` using the `_.identity` function (one that just returns its argument) as an example:

```
_.map({a: 1, b: 2}, _.identity);
//=> [1,2]
```

It would seem that `_.map` only deals with the value parts of the key/value pair, but this limitation is only a matter of use. If we want to deal with the key/value pairs, then we supply a function that expects them:

```
_.map({a: 1, b: 2}, function(v,k) {
  return [k,v];
});
//=> [['a', 1], ['b', 2]]
```

In the spirit of completeness, it's worth noting that the function `_.map` can also take a third argument, the collection itself:

```
_.map({a: 1, b: 2}, function(v,k,coll) {
  return [k, v, _.keys(coll)];
 });
//=> [['a', 1, ['a', 'b']], ['b', 2, ['a', 'b']]]
```

The point of a collection-centric view, as advocated by Underscore and functional programming in general, is to establish a consistent processing idiom so that we can reuse a comprehensive set of functions. As the great luminary Alan Perlis once stated:

> It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.

Throughout this book, I emphasize the notion of empowering our data through the use of generic functions built on a collection-centric philosophy.

Other Examples of Applicative Programming

To close out my discussion of applicative programming, I offer examples illustrating it, with some dialog along the way.

reduceRight

You've already seen the `_.reduce` function, but I failed to mention its sibling `_.reduceRight`. The two functions operate in much the same way, except that `_.reduce` works from left to right, whereas `_.reduceRight` works from right to left. Observe the differences:

```
var nums = [100,2,25];

function div(x,y) { return x/y };

_.reduce(nums, div);
//=> 2

_.reduceRight(nums, div);
//=> 0.125
```

The work of `_.reduce` is similar to `(100/2) / 25` while `_.reduceRight` is `(25/2) / 100`. If the function supplied to the reduce siblings is associative, then they wind up returning the same values, but otherwise the difference in ordering can prove useful. Many common functions can be created using `_.reduceRight`. Here are a couple more examples:

```
function allOf(/* funs */) {
  return _.reduceRight(arguments, function(truth, f) {
    return truth && f();
  }, true);
}

function anyOf(/* funs */) {
  return _.reduceRight(arguments, function(truth, f) {
    return truth || f();
  }, false);
}
```

Example usages of `allOf` and `anyOf` are as follows:

```
function T() { return true }
function F() { return false }

allOf();
//=> true
allOf(T, T);
//=> true
allOf(T, T, T , T , F);
//=> false
```

```
anyOf(T, T, F);
//=> true
anyOf(F, F, F, F);
//=> false
anyOf();
//=> false
```

The `_.reduceRight` function has further advantages in languages providing lazy evaluation, but since JavaScript is not such a language, evaluation *order* is the key factor (Bird 1988).[5]

find

The `find` function is fairly straightforward; it takes a collection and a predicate and returns the first element for which the predicate returns `true`. An example of `find` is as follows:

```
_.find(['a', 'b', 3, 'd'], _.isNumber);
//=> 3
```

Notice the use of the built-in function `_.isNumber` as the predicate function. Underscore comes with numerous predicates ready for use, including `_.isEqual`, `_.isEmpty`, `_.isElement`, `_.isArray`, `_.isObject`, `_.isArguments`, `_.isFunction`, `_.isString`, `_.isNumber`, `_.isFinite`, `_.isBoolean`, `_.isDate`, `_.isRegExp`, `_.isNaN`, `_.isNull`, and `_.isUndefined`. I will use some or all of them over the course of this book.

reject

Underscore's `_.reject` is essentially the opposite of `_.filter`; it takes a predicate and returns a collection of values that excludes values for which the predicate returned `true`. For example:

```
_.reject(['a', 'b', 3, 'd'], _.isNumber);
//=> ['a', 'b', 'd']
```

This is the same as reversing the truthiness of the predicate to `_.filter`. In fact, a simple function called `complement` would perform just such a task:[6]

```
function complement(pred) {
  return function() {
    return !pred.apply(null, _.toArray(arguments));
  };
}
```

5. The `allOf` and `anyOf` functions could have just as easily been written using Underscore's `reduce`, but I chose to use the former for the purpose of illustrating `reduceRight`.

6. Passing `null` as the first argument to `apply` is worth a mention. Recall that the first argument to `apply` is the "target" object setting the `this` reference inside the called function. Since I can't know what the target object should be, or even if it's needed at all, I use `null` to signal that `this` should just refer to the global object.

The `complement` function takes a predicate and returns a function that reverses the sense of the result of said predicate. It can then be used with `_.filter` to achieve the same effect as `_.reject`:

```
_.filter(['a', 'b', 3, 'd'], complement(_.isNumber));
//=> ['a', 'b', 'd']
```

The `complement` function is an example of a higher-order function. Although I touched briefly on what that means earlier, I will defer a deeper discussion until Chapter 3.

all

The `_.all` function takes a collection and a predicate, and returns `true` if *all* of the elements within return `true` on the predicate check. For example:

```
_.all([1, 2, 3, 4], _.isNumber);
//=> true
```

Of course, if *any* of the elements fail the predicate test, then `_.all` returns `false`.

any

The `_.any` function takes a collection and a predicate, and returns `true` if *any* of the elements within return `true` on the predicate check. For example:

```
_.any([1, 2, 'c', 4], _.isString);
//=> true
```

Of course, if *all* of the elements fail the predicate test, then `_.any` returns `false`.

sortBy, groupBy, and countBy

The last three applicative functions that I'll discuss are related, in that they all perform some action based on the result of a given criteria function. The first of the three, `_.sortBy`, takes a collection and a function, and returns a sorted collection based on the criteria determined by the passed function. For example:

```
var people = [{name: "Rick", age: 30}, {name: "Jaka", age: 24}];

_.sortBy(people, function(p) { return p.age });

//=> [{name: "Jaka", age: 24}, {name: "Rick", age: 30}]
```

The `_.groupBy` function takes a collection and a criteria function, and returns an object where the keys are the criteria points returned by the function, and their associated values are the elements that matched. For example:

```
var albums = [{title: "Sabbath Bloody Sabbath", genre: "Metal"},
              {title: "Scientist", genre: "Dub"},
              {title: "Undertow", genre: "Metal"}];

_.groupBy(albums, function(a) { return a.genre });
```

```
//=> {Metal:[{title:"Sabbath Bloody Sabbath", genre:"Metal"},
//           {title:"Undertow", genre:"Metal"}],
//    Dub:  [{title:"Scientist", genre:"Dub"}]}
```

The `_.groupBy` function is extremely handy, and will show up numerous times throughout the course of this book.

The final applicative function I'll discuss is called `_.countBy`. This function works similarly to `_.groupBy`, except that it returns an object with keys of the match criteria associated with its count, as shown in the following:

```
_.countBy(albums, function(a) { return a.genre });

//=> {Metal: 2, Dub: 1}
```

That wraps up this discussion of applicative functional programming. I started with the most common case of a functional style that you're likely to encounter in JavaScript code, so that you have some background knowledge before venturing deeper into the wilderness. Up next, I'll cover one more common topic in JavaScript code: closures.

Defining a Few Applicative Functions

I've shown many of the applicative functions offered by Underscore, but what about creating some of your own? The process is fairly straightforward: define a function that takes a function and then calls it.

A simple function that takes some number of arguments and concatenates them is *not* applicative:

```
function cat() {
  var head = _.first(arguments);
  if (existy(head))
    return head.concat.apply(head, _.rest(arguments));
  else
    return [];
}

cat([1,2,3], [4,5], [6,7,8]);
//=> [1, 2, 3, 4, 5, 6, 7, 8]
```

While considerably useful, `cat` doesn't expect to receive any functions as arguments.[7] Likewise, a function `construct` that takes an element and an array and places the element in the front of the array may use `cat`:

```
function construct(head, tail) {
  return cat([head], _.toArray(tail));
}
```

7. The `cat` function might receive functions in the arrays that it takes, but that is tangential to the point.

```
construct(42, [1,2,3]);
//=> [42, 1, 2, 3]
```

While `construct` uses `cat` within its body, it does not receive it as an argument, so it fails the applicative test.

Instead, a function `mapcat`, defined as follows, is applicative:

```
function mapcat(fun, coll) {
  return cat.apply(null, _.map(coll, fun));
}
```

The function `mapcat` does indeed take a function, `fun`, that it uses in the same manner as `_.map`, calling it for every element in the given collection. This use of `fun` is the applicative nature of `mapcat`. Additionally, `mapcat` concatenates all of the elements of the result of `_.map`:

```
mapcat(function(e) {
  return construct(e, [","]);
}, [1,2,3]);
//=> [1, ",", 2, ",", 3, ","]
```

The operation of `mapcat` is such that when the mapped function returns an array, it can be flattened a level. We could then use `mapcat` and another function, `butLast`, to define a third function, `interpose`:

```
function butLast(coll) {
  return _.toArray(coll).slice(0, -1);
}

function interpose (inter, coll) {
  return butLast(mapcat(function(e) {
    return construct(e, [inter]);
  },
  coll));
}
```

Using `interpose` is straightforward:

```
interpose(",", [1,2,3]);

//=> [1, ",", 2, ",", 3]
```

This is a key facet of functional programming: the gradual definition and use of discrete functionality built from lower-level functions. Very often you will see (and in this book I will preach the case vociferously) a chain of functions called one after the other, each gradually transforming the result from the last to reach a final solution.

Data Thinking

Throughout this book, I'll take the approach of using minimal data types to represent abstractions, from sets to trees to tables. In JavaScript, however, although its object types are extremely powerful, the tools provided to work with them are not entirely functional. Instead, the larger usage pattern associated with JavaScript objects is to attach methods for the purposes of polymorphic dispatch. Thankfully, you can also view an unnamed (not built via a constructor function) JavaScript object as simply an associative data store.[8]

If the only operations that we can perform on a `Book` object or an instance of an `Employee` type are `setTitle` or `getSSN`, then we've locked our data up into per-piece-of-information micro-languages (Hickey 2011). A more flexible approach to modeling data is an associative data technique. JavaScript objects, even minus the prototype machinery, are ideal vehicles for associative data modeling, where named values can be structured to form higher-level data models, accessed in uniform ways.[9]

Although the tools for manipulating and accessing JavaScript objects as data maps are sparse within JavaScript itself, thankfully Underscore provides a bevy of useful operations. Among the simplest functions to grasp are `_.keys`, `_.values`, and `_.pluck`. Both `_.keys` and `_.values` are named according to their functionality, which is to take an object and return an array of its keys or values:

```
var zombie = {name: "Bub", film: "Day of the Dead"};

_.keys(zombie);
//=> ["name", "film"]

_.values(zombie);
//=> ["Bub", "Day of the Dead"]
```

The `_.pluck` function takes an array of objects and a string and returns all of the values at the given key for each object in the array:

```
_.pluck([{title: "Chthon", author: "Anthony"},
         {title: "Grendel", author: "Gardner"},
         {title: "After Dark"}],
        'author');

//=> ["Anthony", "Gardner", undefined]
```

8. There has been some discussion within the ECMAScript effort to provide simple map (and set) types, divorced from the prototype system. More information is found at *http://wiki.ecmascript.org/doku.php?id=harmony:simple_maps_and_sets*.

9. JavaScript's ability to provide uniform access across its associative data types is a boon in allowing you to write a powerful suite of functions for manipulating data generically. JavaScript's `for…in` loop and the indexed access operator form the basis for much of Underscore.

All three of these functions deconstruct the given objects into arrays, allowing you to perform sequential actions. Another way of viewing a JavaScript object is as an array of arrays, each holding a key and a value. Underscore provides a function named `_.pairs` that takes an object and turns it into this nested array:

```
_.pairs(zombie);

//=> [["name", "Bub"], ["film", "Day of the Dead"]]
```

This nested array view can be processed using sequential operations and reassembled into a new object using Underscore's `_.object` function:

```
_.object(_.map(_.pairs(zombie), function(pair) {
  return [pair[0].toUpperCase(), pair[1]];
}));

//=> {FILM: "Day of the Dead", NAME: "Bub"};
```

Aside from changing the key in some subtle way, another common function on maps is to flip the keys and values via the `_.invert` function:

```
_.invert(zombie);
//=> {"Bub": "name", "Day of the Dead": "film"}
```

It's worth mentioning that unlike in many other languages, JavaScript object keys can only ever be strings. This may occasionally cause confusion when using `_.invert`:

```
_.keys(_.invert({a: 138, b: 9}));
//=> ['9', '138']
```

Underscore also provides functions for filling in and removing values from objects according to the values that they take:

```
_.pluck(_.map([{title: "Chthon", author: "Anthony"},
               {title: "Grendel", author: "Gardner"},
               {title: "After Dark"}],
              function(obj) {
                return _.defaults(obj, {author: "Unknown"})
              }),
        'author');

//=> ["Anthony", "Gardner", "Unknown"]
```

In this example, each object is preprocessed with the `_.defaults` function to ensure that the `author` field contains a useful value (rather than `undefined`). While `_.defaults` augments incoming objects, two functions—`_.pick` and `_.omit`—(potentially) filter objects based on their arguments:

```
var person = {name: "Romy", token: "j3983ij", password: "tigress"};

var info = _.omit(person, 'token', 'password');
info;
//=> {name: "Romy"}
```

```
var creds = _.pick(person, 'token', 'password');
creds;

//=> {password: "tigress", token: "j3983ij"};
```

Using the same "dangerous" keys, `token` and `password`, shows that the `_.omit` function takes a blacklist to remove keys from an object, and `_.pick` takes a whitelist to take keys (both nondestructively).

Finally, Underscore provides selector functions useful in finding certain objects based on keyed criteria, `_.findWhere` and `_.where`. The `_.findWhere` function takes an array of objects and returns the first one that matches the criteria given in the object in the second argument:

```
var library = [{title: "SICP", isbn: "0262010771", ed: 1},
               {title: "SICP", isbn: "0262510871", ed: 2},
               {title: "Joy of Clojure", isbn: "1935182641", ed: 1}];

_.findWhere(library, {title: "SICP", ed: 2});

//=> {title: "SICP", isbn: "0262510871", ed: 2}
```

The `_.where` function operates similarly except that it operates over an array and returns *all* of the objects that match the criteria:

```
_.where(library, {title: "SICP"});

//=> [{title: "SICP", isbn: "0262010771", ed: 1},
//    {title: "SICP", isbn: "0262510871", ed: 2}]
```

This type of usage pattern points to a very important data abstraction: the table. In fact, using Underscore's object manipulation functions, you can derive an experience very similar to that of SQL, where logical data tables are filtered and processed according to a powerful declarative specification. However, as I'll show next, to achieve a more fluent table processing API, I'll need to step it up beyond what Underscore provides and take advantage of functional techniques. The functions created in this section implement a subset of the relational algebra on which all SQL engines are built (Date 2003). I will not dive deeply into the relational algebra, but will instead work at the level of a pseudo-SQL. I assume a base-level proficiency in SQL-like languages.

"Table-Like" Data

Table 2-1 presents one way to look at the data in the `library` array.

Table 2-1. A data table view of an array of JavaScript objects

title	isbn	ed
SICP	0262010771	1
SICP	0262510871	2
Joy of Clojure	1935182641	1

Here, each row is equivalent to a JavaScript object and each cell is a key/value pair in each object. The information in Table 2-2 corresponds to an SQL query of the form `SELECT title FROM library`.

Table 2-2. A table of the titles

title
SICP
SICP
Joy of Clojure

A way to achieve the same effect using the tools that I've explored so far is as follows:

```
_.pluck(library, 'title');

//=> ["SICP", "SICP", "Joy of Clojure"]
```

The problem is that the result from the `_.pluck` function is of a different abstraction than the table abstraction. While technically an array of strings *is* an array of objects, the abstraction is broken using `_.pluck`. Instead, you need a function that allows a similar capability to the SQL's `SELECT` statement, while preserving the table abstraction. The function `project` will serve as the stand-in for `SELECT`:[10]

```
function project(table, keys) {
  return _.map(table, function(obj) {
    return _.pick.apply(null, construct(obj, keys));
  });
};
```

The `project` function uses the `_.pick` function on each object in the array to pull out whitelisted keys into new objects, thus preserving the table abstraction:

```
var editionResults = project(library, ['title', 'isbn']);

editionResults;
//=> [{isbn: "0262010771", title: "SICP"},
//    {isbn: "0262510871", title: "SICP"},
//    {isbn: "1935182641", title: "Joy of Clojure"}];
```

10. It's an odd mistake of history that the traditional SQL `SELECT` is actually the `PROJECT` statement in relational algebra. I'll use `project` for now because Underscore already provides `select` as an alias for `filter`.

As shown, the `project` function itself returns a table-like data structure, which can be further processed using `project`:

```
var isbnResults = project(editionResults, ['isbn']);

isbnResults;
//=> [{isbn: "0262010771"},{isbn: "0262510871"},{isbn: "1935182641"}]
```

Finally, the abstraction can be intentionally broken by purposefully pulling out only the desired data:

```
_.pluck(isbnResults, 'isbn');
//=> ["0262010771", "0262510871", "1935182641"]
```

This intentional extraction is a deliberate act to "hand over" data from one module or function to the next. While `project` works on the table abstraction, another fictional function, `populateISBNSelectBox`, would probably work with arrays of strings that might then construct DOM elements of the form `<option value="1935182641">1935182641</option>`. Functional programmers think deeply about their data, the transformations occurring on it, and the hand-over formats between the layers of their applications. Visually, you can picture the high-level data-centric thinking as in Figure 2-1 (Gruber 2004).

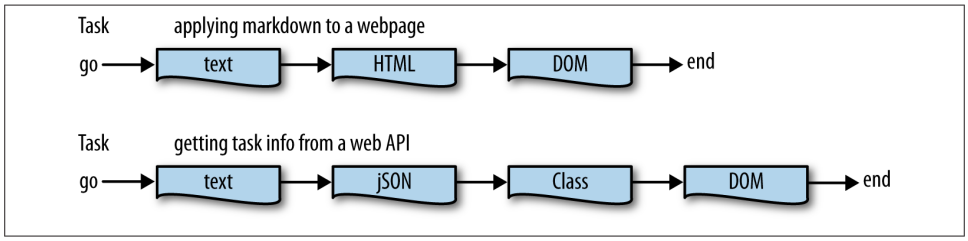

Figure 2-1. Data transformations can be used to abstract tasks

Let's explore this table abstraction just a little bit more before diving deeper into data transformation and hand overs. For example, most SQL engines provide an `AS` statement used to alias column names. In SQL, the `AS` looks like the following:

```
SELECT ed AS edition FROM library;
```

The preceding query would output the results shown in Table 2-3:

Table 2-3. A table of the aliased editions

edition
1
2
1

However, before I implement `as` to work against a table, it would behoove me to create a utility function, `rename`, that renames keys based on a given renaming criteria map:

```
function rename(obj, newNames) {
  return _.reduce(newNames, function(o, nu, old) {
    if (_.has(obj, old)) {
      o[nu] = obj[old];
      return o;
    }
    else
      return o;
  },
  _.omit.apply(null, construct(obj, _.keys(newNames))));
};
```

One important point about the implementation of `rename` is that it uses the `_.reduce` function to reconstruct an object using Underscore's alternative mode for traversing over the key/value pairings that preserves the "mappiness" of the accumulator. I take advantage of this fact by renaming keys via direct array manipulation, according to the `renaming` map. It will be more clear how this works with an example:

```
rename({a: 1, b: 2}, {'a': 'AAA'});

//=> {AAA: 1, b: 2}
```

I can implement an `as` function using `rename` to work against the table abstraction as follows:

```
function as(table, newNames) {
  return _.map(table, function(obj) {
    return rename(obj, newNames);
  });
};
```

As you'll notice, `as` works against the table abstraction by simply mapping the `rename` over each of the contained objects. Observe:

```
as(library, {ed: 'edition'});

//=> [{title: "SICP", isbn: "0262010771", edition: 1},
//    {title: "SICP", isbn: "0262510871", edition: 2},
//    {title: "Joy of Clojure", isbn: "1935182641", edition: 1}]
```

Because both `as` and `project` work against the same abstraction, I can chain the calls together to produce a new table like that given by the aforementioned SQL statement:

```
project(as(library, {ed: 'edition'}), ['edition']);

//=> [{edition: 1}, {edition: 2}, {edition: 1}];
```

Finally, I can square the circle of providing basic SQL capabilities against a table abstraction by implementing a function akin to SQL's `WHERE` clause, named `restrict` (Date 2011):

```
function restrict(table, pred) {
  return _.reduce(table, function(newTable, obj) {
    if (truthy(pred(obj)))
      return newTable;
    else
      return _.without(newTable, obj);
  }, table);
};
```

The `restrict` function takes a function that acts as a predicate against each object in the table. Whenever the predicate returns a falsey value, the object is disregarded in the final table. Here's how `restrict` can work to remove all first editions:

```
restrict(library, function(book) {
  return book.ed > 1;
});

//=> [{title: "SICP", isbn: "0262510871", ed: 2}]
```

And like the rest of the functions that work against the table abstraction, `restrict` can be chained:

```
restrict(
  project(
    as(library, {ed: 'edition'}),
    ['title', 'isbn', 'edition']),
  function(book) {
    return book.edition > 1;
});

//=> [{title: "SICP", isbn: "0262510871", edition: 2},]
```

An equivalent SQL statement could be written as follows:

```
SELECT title, isbn, edition FROM (
  SELECT ed AS edition FROM library
) EDS
WHERE edition > 1;
```

Although they're not as attractive as the equivalent SQL, the functions `project`, `as`, and `restrict` work against a common table abstraction—a simple array of objects. This is data thinking.

Summary

This chapter focused on first-class functions. First-class functions are functions that can be treated like any other piece of data:

- They can be stored in a variable.
- They can be stored in an array slot.

- They can be stored in an object field.
- They can be created as needed.
- They can be passed to other functions.
- They can be returned from functions.

That JavaScript supports first-class functions is a great boon to practicing functional programming. One particular form of functional programming—and one that most readers will be familiar with—is known as *applicative programming*. Examples of functions that allow applicative programming such as `_.map`, `_.reduce`, and `_.filter` were shown, and new applicative functions were created later.

What makes applicative programming particularly powerful is a focus in most JavaScript applications on dealing with collections of data, whether they're arrays, objects, arrays of objects, or objects of arrays. A focus on fundamental collection types allowed us to build a set of SQL-like relational operators working against a simple "table" abstraction built from arrays of objects.

The next chapter is a transition chapter to cover the fundamental topic of variable scope and closures.