CHAPTER 8

# Flow-Based Programming

This chapter continues the discussion of functional style by showing how functions, together with purity and isolated change, can compose to offer a fairly fluent programming style. The idea of snapping functional blocks together will be discussed herein and demonstrated with relevant examples.

## Chaining

If you recall, in the implementation of `condition1` from Chapter 5, I resorted to using the following lines of code:

```
// ...
    var errors = mapcat(function(isValid) {
      return isValid(arg) ? [] : [isValid.message];
    }, validators);
// ...
```

The reason for this bit of trickery was that while the final result needed to be an array of error strings, each intermediate step could be either an array of suberror messages or nothing at all. Another reason was that I wanted to combine disparate behaviors, each with different return types. It would be much easier to compose these behaviors if the return value of one was of a form agreeable to the input arguments to the other. Take, for example, the following code:

```
function createPerson() {
  var firstName = "";
  var lastName = "";
  var age = 0;

  return {
    setFirstName: function(fn) {
      firstName = fn;
      return this;
    },
```

```
    setLastName: function(ln) {
      lastName = ln;
      return this;
    },
    setAge: function(a) {
      age = a;
      return this;
    },
    toString: function() {
      return [firstName, lastName, age].join(' ');
    }
  };
}

createPerson()
  .setFirstName("Mike")
  .setLastName("Fogus")
  .setAge(108)
  .toString();

//=> "Mike Fogus 108"
```

The "magic" that allows method chains is that each method in the chain returns the same host object reference (Stefanov 2010). The chaining of methods via common return value is effectively a design pattern in JavaScript finding its way into jQuery and even Underscore. In fact, three useful functions in Underscore are `_.tap`, `_.chain`, and `_.value`. If you recall from Chapter 2, I used these functions to implement the `lyricSegment` function used to build a "99 bottles" song generator. However, in that implementation I glossed over just how these functions operate.

The `_.chain` function is the most hardcore of the three, allowing you to specify an object as an implicit target to repeated Underscore functions pretending to be methods. A simple example works best to start understanding `_.chain`:

```
_.chain(library)
 .pluck('title')
 .sort();

//=> _
```

Um. What?[1]

Thankfully, there is a good explanation for why the Underscore object was returned. You see, the `_.chain` function takes some object and wraps it in another object that contains modified versions of all of Underscore's functions. That is, where a function like `_.pluck` has a call signature like `function pluck(array, propertyName)` by

1. If you're using a minified version of Underscore, you might actually see a differently named object here. That is only the result of renaming by the chosen minification tool.

default, the modified version found in the wrapper object used by `_.chain` looks like `function pluck(propertyName)`. Underscore uses a lot of interesting trickery to allow this to happen, but the result is that what passes from one wrapper method call to the next is the wrapper object and *not* the target object itself. Therefore, whenever you want to end the call to `_.chain` and extract the final value, the `_.value` function is used:

```
_.chain(library)
 .pluck('title')
 .sort()
 .value();

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

With the use of `_.result`, you take a value from the world of the wrapper object and bring it into the "real world." This notion will pop up again a couple of sections from now. When using the `_.chain` function, you might receive results for any number of buggy reasons. Imagine the following scenario:

```
var TITLE_KEY = 'titel';

// ... a whole bunch of code later

_.chain(library)
 .pluck(TITLE_KEY)
 .sort()
 .value();

//=> [undefined, undefined, undefined]
```

Because the code is compact, the problem is obvious—I misspelled "title." However, in a large codebase you're likely to start debugging closer to the point of failure. Unfortunately, with the presence of the `_.chain` call, there is seemingly no easy way to *tap into* the chain to inspect intermediate values. Not so. In fact, Underscore provides a `_.tap` function that, given an object and a function, calls the function with the object and returns the object:

```
_.tap([1,2,3], note);
;; NOTE: 1,2,3
//=> [1, 2, 3]
```

Passing the `note` function[2] to Underscore's `tap` shows that indeed the function is called and the array returned. As you might suspect, `_.tap` is also available in the wrapper object used by `_.chain`, and therefore can be used to inspect intermediate values, like so:

```
_.chain(library)
 .tap(function(o) {console.log(o)})
 .pluck(TITLE_KEY)
 .sort()
```

2. The `note` function was defined all the way back in Chapter 1.

```
  .value();

// [{title: "SICP" ...
//=> [undefined, undefined, undefined]
```

Nothing seems amiss in the form of the `library` table, but what about if I move the tap to a different location:

```
_.chain(library)
 .pluck(TITLE_KEY)
 .tap(note)
 .sort()
 .value();

// NOTE:  ,,
//=> [undefined, undefined, undefined]
```

Now wait a minute; the result of the pluck is an odd looking array. At this point, the tap has pointed to the location of the problem: the call to `_.pluck`. Either there is a problem with `TITLE_KEY` or a problem with `_.pluck` itself. Thankfully, the problem lies in the code under my control.

The use of `_.chain` is very powerful, especially when you want to fluently describe a sequence of actions occurring on a single target. However, there is one limitation of `_chain`—it's not lazy. What I mean by what is hinted at in the following code:

```
_.chain(library)
 .pluck('title')
 .tap(note)
 .sort();

// NOTE: SICP,SICP,Joy of Clojure
//=> _
```

Even though I never explicitly asked for the wrapped value with the `_.value` function, all of the calls in the chain were executed anyway. If `_.chain` were lazy, then none of the calls would have occurred *until* the call to `_.value`.

## A Lazy Chain

Taking a lesson from the implementation of `trampoline` from Chapter 6, I can implement a lazy variant of `_.chain` that will not run any target methods until a variant of `_.value` is called:

```
function LazyChain(obj) {
  this._calls  = [];
  this._target = obj;
}
```

The constructor for the `LazyChain` object is simple enough; it takes a target object like `_.chain` and sets up an empty call array. While the operation of `trampoline` from Chapter 6 operated on an implicit chain of calls, `LazyChain` works with an explicit array of…something. However, the question remains as to what it is that I put into the `_calls` array. The most logical choice is, of course, functions, as shown below:

```
LazyChain.prototype.invoke = function(methodName /*, args */) {
  var args = _.rest(arguments);

  this._calls.push(function(target) {
    var meth = target[methodName];

    return meth.apply(target, args);
  });

  return this;
};
```

The `LazyChain#invoke` method is fairly straightforward, but I could stand to walk through it. The arguments to `LazyChain#invoke` are a method name in the form of a string, and any additional arguments to the method. What `LazyChain#invoke` does is to "wrap" up the actual method call in a closure and push it onto the `_calls` array. Observe what the `_calls` array looks like after a single invocation of `LazyChain#invoke` here:

```
new LazyChain([2,1,3]).invoke('sort')._calls;
//=> [function (target) { ... }]
```

As shown, the only element in the `_calls` array after adding one link to the lazy chain is a single function that corresponds to a deferred `Array#sort` method call on the array `[2,1,3]`.

A function that wraps some behavior for later execution is typically called a *thunk*[3] (see Figure 8-1). The thunk that's stored in `_calls` expects some intermediate target that will serve as the object receiving the eventual method call.

3. The term "thunk" has roots extending all the way back to ALGOL.

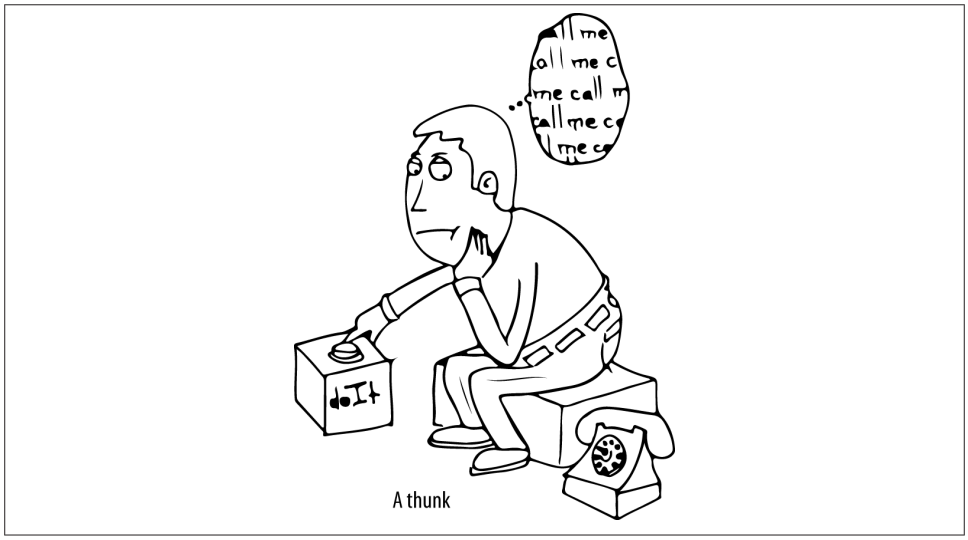


*Figure 8-1. A thunk is a function waiting to be called*

While thunks are not always functions in every programming language that supports them, in JavaScript it makes sense to implement them as such because functions are readily available.

Since the thunk is waiting to be called, why don't I just call it to see what happens:

```
new LazyChain([2,1,3]).invoke('sort')._calls[0]();

// TypeError: Cannot read property 'sort' of undefined
```

Well, that was less than satisfying. The problem is that directly calling the thunk is not enough to make it execute properly. If you'll recall, the thunk expected a `target` object to execute its closed-over method on. To make it work, I would need to somehow pass the original array as an argument to the thunk, as shown in Figure 8-2.

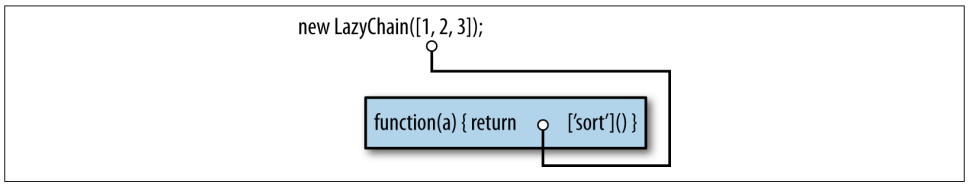


*Figure 8-2. To make the LazyChain work, I have to find a way to loop the original object back into the call*

I could satisfy the argument lookback manually by pasting the array into the thunk call as in the following:

```
new LazyChain([2,1,3]).invoke('sort')._calls[0]([2,1,3]);

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

Placing the argument directly into the thunk call seems not only like cheating, but also like a terrible API. Instead, I can use the `_.reduce` function to provide the loopback argument not only to the initial thunk, but also every intermediate call on the `_calls` array:

```
LazyChain.prototype.force = function() {
  return _.reduce(this._calls, function(target, thunk) {
    return thunk(target);
  }, this._target);
};
```

The `LazyChain#force` function is the execution engine for the lazy chaining logic. As shown in Figure 8-3, the use of `_.reduce` nicely provides the same kind of trampolining logic as demonstrated in Chapter 6. Starting with the initial target object, the thunk calls are called, one by one, with the result of the previous call.

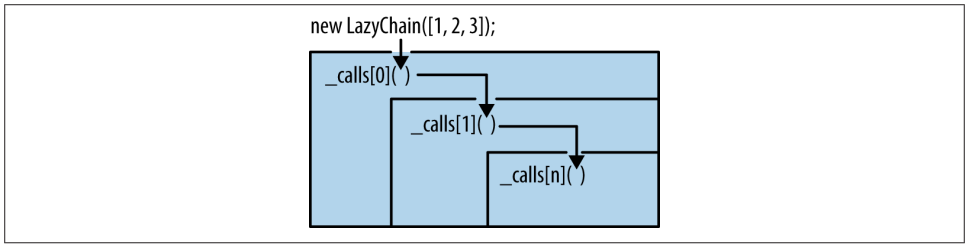


*Figure 8-3. Using reduce allows me to pass the intermediate result forward into each thunk*

Now that `LazyChain#force` is in place, observe what happens when it is used to "terminate" a lazy chain:

```
new LazyChain([2,1,3]).invoke('sort').force();

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

Excellent! The logic seems sound, but what happens when more links are added to the chain? Observe:

```
new LazyChain([2,1,3])
  .invoke('concat', [8,5,7,6])
  .invoke('sort')
  .invoke('join', ' ')
  .force();

//=> "1 2 3 5 6 7 8"
```

I can chain as long as I want while remaining mindful of the way that the types change from one link to the next. I mentioned earlier that `LazyChain` was lazy in its execution. While you might see how that is indeed the case because thunks are stored in the `_calls` array and never executed until `LazyChain#force`, it's still better to show it actually being lazy. First, let me implement a lazy version of `_.tap` that works with `LazyChain` instances:

```
LazyChain.prototype.tap = function(fun) {
  this._calls.push(function(target) {
    fun(target);
    return target;
  });

  return this;
}
```

The operation of `LazyChain#tap` is similar to `LazyChain#invoke` because the actual work (i.e., calling a function and returning the target) is wrapped in a thunk. I show how tap works below:

```
new LazyChain([2,1,3])
  .invoke('sort')
  .tap(alert)
  .force();

// alert box pops up
//=> "1,2,3"
```

But what happens if I never call `LazyChain#force`?

```
var deferredSort = new LazyChain([2,1,3])
  .invoke('sort')
  .tap(alert);

deferredSort;
//=> LazyChain
```

Nothing happens! I can hold onto `deferredSort` as long as I want and it'll never execute until I explicitly invoke it:

```
// ... in the not too distant future

deferredSort.force();

// alert box pops up
//=> [1, 2, 3]
```

This operation is very similar to the way that something like jQuery promises work. Before I talk a little bit about promises, however, I want to explore an easy extension to `LazyChain` that allows me to, well, chain lazy chains to other lazy chains. That is, keeping in mind that at the heart of a `LazyChain` is just an array of thunks, I can change the constructor to concatenate the arrays when presented with another `LazyChain` as its argument:

```
function LazyChainChainChain(obj) {
  var isLC = (obj instanceof LazyChain);

  this._calls  = isLC ? cat(obj._calls, []) : [];
  this._target = isLC ? obj._target : obj;
}

LazyChainChainChain.prototype = LazyChain.prototype;
```

That is, if the argument to the constructor is another `LazyChain` instance, then just steal its call chain and target object. Observe the chaining of chains:

```
new LazyChainChainChain(deferredSort)
  .invoke('toString')
  .force();

// an alert box pops up
//=> "1,2,3"
```

Allowing chains to compose in this way is a very powerful idea. It allows you to build up a library of discrete behaviors without worrying about the final result. There are other ways to enhance `LazyChain`, such as caching the result and providing an interface that does not rely on strings, but I leave that as an exercise for the reader.

## Promises

While `LazyChain` and `LazyChainChainChain` are useful for packaging the description of a computation for later execution, jQuery[4] provides something called a *promise* that works similarly, but slightly differently. That is, jQuery promises are intended to provide a fluent API for sequencing asynchronous operations that run concurrent to the main program logic.

First, the simplest way to look at a promise is that it represents an unfulfilled activity. As shown in the following code, jQuery allows the creation of promises via `$.Deferred`:

```
var longing = $.Deferred();
```

I can now grab a promise from the `Deferred`:

```
longing.promise();
//=> Object
```

The object returned is the handle to the unfulfilled action:

```
longing.promise().state();
//=> "pending"
```

4. Other JavaScript libraries that offer promises similar to jQuery's include, but are not limited to: Q, RSVP.js, when.js, and node-promises.

As shown, the promise is in a holding pattern. The reason for this is of course that the promise was never fulfilled. I can do so simply by resolving it:

```
longing.resolve("<3");

longing.promise().state();
//=> "resolved"
```

At this point, the promise has been fulfilled and the value is accessible:

```
longing.promise().done(note);
// NOTE: <3
//=> <the promise itself>
```

The `Deferred#done` method is just one of many useful chaining methods available in the promise API. I will not go into depth about jQuery promises, but a more complicated example could help to show how they differ from lazy chains. One way to build a promise in jQuery is to use the `$.when` function to start a promise chain, as shown here:

```
function go() {
  var d = $.Deferred();

  $.when("")
   .then(function() {
     setTimeout(function() {
       console.log("sub-task 1");
     }, 5000)
   })
   .then(function() {
     setTimeout(function() {
       console.log("sub-task 2");
     }, 10000)
   })
   .then(function() {
     setTimeout(function() {
       d.resolve("done done done done");
     }, 15000)
   })

  return d.promise();
}
```

The promise chain built in the `go` function is very simple-minded. That is, all I've done is to tell jQuery to kick off three asynchronous tasks, each delayed by increasingly longer timings. The `Deferred#then` methods each take a function and execute them immediately. Only in the longest-running task is the `Deferred` instance resolved. Running `go` illustrates this example:

```
var yearning = go().done(note);
```

I tacked on a `done` call to the promise that will get called whenever the promise is resolved. However, immediately after running `go`, nothing appears to have happened.

That's because due to the timeouts in the subtasks, the console logging has not yet occurred. I can check the promise state using the aptly named `state` method:

```
yearning.state();
//=> "pending"
```

As you might expect, the state is still pending. After a few seconds, however:

```
// (console) sub-task 1
```

The timeout of the first subtask triggers and a notification is printed to the console.

```
yearning.state();
//=> "pending"
```

Of course, because the other two actions in the original promise chain are awaiting timeouts, the state is still pending. However, again waiting for some number of seconds to pass shows the following:

```
// (console) sub-task 2

// ... ~5 seconds later

// NOTE: done done done done
```

Eventually, the final link in the deferred chain is called, and the done notification is printed by the `note` function. Checking the state one more time reveals the following:

```
yearning.state();
//=> "resolved"
```

Of course, the promise has been resolved because the final subtask ran and called `resolve` on the original `Deferred` instance. This sequence of events is very different from that presented using `LazyChain`. That is, a `LazyChain` represents a strict sequence of calls that calculate a value. Promises, on the other hand, also represent a sequence of calls, but differ in that once they are executed, the value is available on demand.

Further, jQuery's promise API is meant to define aggregate tasks composed of some number of asynchronous subtasks. The subtasks themselves execute, as possible, concurrently. However, the aggregate task is not considered completed until every subtask has finished *and* a value is delivered to the promise via the `resolve` method.

A lazy chain also represents an aggregate task composed of subtasks, but they, once forced, are always run one after the other. The difference between the two can be summarized as the difference between aggregating highly connected tasks (`LazyChain`) versus loosely related (`Deferred`) tasks.

Most of jQuery's asynchronous API calls now return promises, so the result of an async call is chainable according to the promise API. However, the complete specification of this API is outside the scope of this book.

## Pipelining

Chaining is a useful pattern for creating fluent APIs built around objects and method calls, but is less useful for functional APIs. The Underscore library is built for chaining via `_.chain`, as most functions take a collection as the first argument. By contrast, the functions in this book take functions as their first argument. This choice was explicit, to foster partial application and currying.

There are various downsides to method chaining including tight-coupling of object set and get logic (which Fowler refers to as command/query separation [2010]) and awkward naming problems. However, the primary problem is that very often method chains mutate some common reference from one call to the rest, as shown in Figure 8-4. Functional APIs, on the other hand, work with values rather than references and often subtly (and sometimes not so subtly) transform the data, returning the new result.

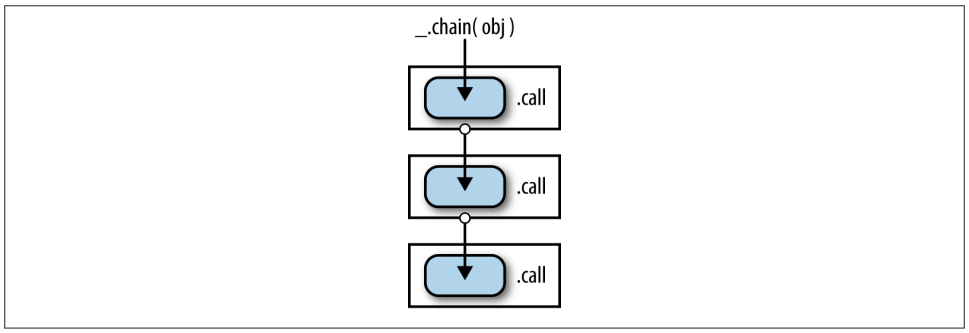


*Figure 8-4. Chained method calls work to mutate a common reference*

In this section, I'll talk about function pipelines and how to use them to great effect. In an ideal world, the original data presented to a function should remain the same after the call. The chain of calls in a functional code base are built from expected data values coming in, nondestructive transformations, and new data returned—strung end to end, as shown in Figure 8-5.

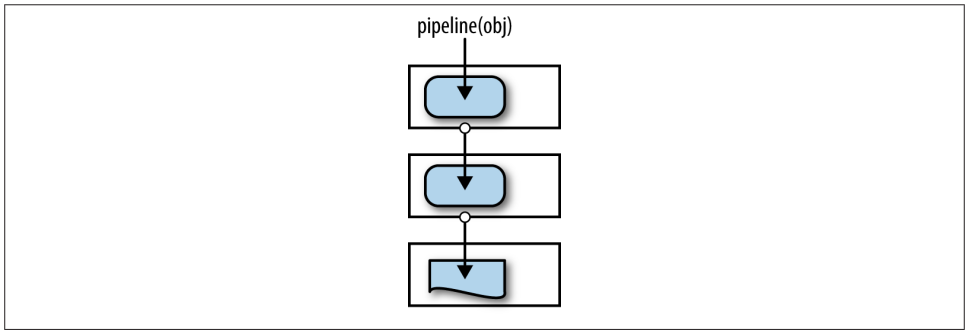


*Figure 8-5. Pipelined functions work to transform data*

A "faux" API for such a pipeline of transformations can look like the following:

```
pipeline([2, 3, null, 1, 42, false]
  , _.compact
  , _.initial
  , _.rest
  , rev);

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

The sequence of this pipeline call could be described as follows:

1. Take the array `[2, 3, null, 1, 42, false]` and pass it to the `_.compact` function.
2. Take the result of `_.compact` and pass it to `_.initial`.
3. Take the result of `_.initial` and pass it to `_.rest`.
4. Take the result of `_.rest` and pass it to `rev`.

In other words, the pipeline looks like the following if written out as nested calls:

```
rev(_.rest(_.initial(_.compact([2, 3, null, 1, 42, false]))));

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

This description should start setting off alarms bells in you brain. That's because this description is almost the same as the description of `LazyChain#force`. The same result/call weaving is prevalent in both algorithms. Therefore, the implementation of `pipeline` should look very similar to `LazyChain#force`, and indeed it is:

```
function pipeline(seed /*, args */) {
  return _.reduce(_.rest(arguments),
                  function(l,r) { return r(l); },
                  seed);
};
```

The use of `_.reduce` makes `pipeline` almost trivial, however, with a seemingly small amount of code comes great power. Before I dig into this power, look at a few examples of `pipeline` in action:

```
pipeline();
//=> undefined

pipeline(42);
//=> 42

pipeline(42, function(n) { return -n });
//=> -42
```

The first argument to `pipeline` serves as the seed value, or in other words, the value that starts as the argument to the first function. The result of each subsequent function call is then fed into the next function until all are exhausted.[5]

Pipelines are somewhat similar to lazy chains, except they are not lazy and they work against values rather than mutable references.[6] Instead, pipelines are more akin to functions created using `_.compose`. The act of making a pipeline lazy is simply the act of encapsulating it within a function (or thunk if you prefer):

```
function fifth(a) {
  return pipeline(a
    , _.rest
    , _.rest
    , _.rest
    , _.rest
    , _.first);
}
```

And now the act of forcing a pipeline is just to feed it a piece of data:

```
fifth([1,2,3,4,5]);
//=> 5
```

A very powerful technique is to use the abstraction built via a pipeline and insert it into another pipeline. They compose thus:

```
function negativeFifth(a) {
  return pipeline(a
    , fifth
    , function(n) { return -n });
}

negativeFifth([1,2,3,4,5,6,7,8,9]);
//=> -5
```

5. If you want to be truly fancy, then you can call `pipeline` by its proper name: the thrush combinator. I'll avoid that temptation, however.
6. I guess, based on these vast differences, you could say that they are not similar at all.

This is interesting as an illustrative example, but it might be more compelling to show how it could be used to create fluent APIs. Recall the implementation of the relational algebra operators `as`, `project`, and `restrict` from Chapter 2. Each function took as its first argument a table that it used to generate a new table, "modified" in some way. These functions seem perfect for use in a pipeline such as one to find all of the first edition books in a table:

```
function firstEditions(table) {
  return pipeline(table
    , function(t) { return as(t, {ed: 'edition'}) }
    , function(t) { return project(t, ['title', 'edition', 'isbn']) }
    , function(t) { return restrict(t, function(book) {
        return book.edition === 1;
      });
    });
}
```

And here's the use of `firstEditions`:

```
firstEditions(library);

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

For processing and extracting elements from the table, the relational operators worked well, but with `pipeline`, I can make it nicer to deal with.

The problem is that the pipeline expects that the functions embedded within take a single argument. Since the relational operators expect two, an adapter function needs to wrap them in order to work within the pipeline. However, the relational operators were designed very specifically to conform to a consistent interface: the table is the first argument and the "change" specification is the second. Taking advantage of this consistency, I can use `curry2` to build curried versions of the relational operators to work toward a more fluent experience:

```
var RQL = {
  select: curry2(project),
  as: curry2(as),
  where: curry2(restrict)
};
```

I've decided to namespace the curried functions inside of an object `RQL` (standing for *relational query language*) and change the names in two of the circumstances to more closely mimic the SQL operators. Now that they are curried, implementing an improved version of `firstEditions` reads more cleanly:

```
function allFirstEditions(table) {
  return pipeline(table
    , RQL.as({ed: 'edition'})
    , RQL.select(['title', 'edition', 'isbn'])
    , RQL.where(function(book) {
```

```
        return book.edition === 1;
    }));
}
```

Aside from being easier to read,[7] the new `allFirstEditions` function will work just as well:

```
allFirstEditions(table);

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

The use of pipelines in JavaScript, coupled with currying and partial application, works to provide a powerful way to compose functions in a fluent manner. In fact, the functions created in this book were designed to work nicely in pipelines. As an added advantage, functional programming focuses on the transformation of data as it flows from one function to the next, but this fact can often be obscured by indirection and deep function nesting. Using a pipeline can work to make the data flow more explicit. However, pipelines are not appropriate in all cases. In fact, I would rarely use a pipeline to perform side-effectful acts like I/O, Ajax calls, or mutations because they very often return nothing of value.

The data going into a pipeline should be the same after the pipeline has completed. This helps ensure that the pipelines are composable. Is there a way that I can compose impure functions along a pipeline-like structure? In the next section, I'll talk a bit about a way to perform side effects in a composable and fluent way, building on the lessons learned while exploring chains and pipelines.

## Data Flow versus Control Flow

In the `lazyChain` example, I separated out the execution specification (via `.invoke`) from the execution logic (via `.force`). Likewise, with the `pipeline` function, I juxtaposed numerous pure functions to achieve the equivalent of a serial processing pipeline. In both the cases of `lazyChain` and `pipeline`, the value moving from one node in the call sequence to the next was stable. Specifically, `lazyChain` always returned some `LazyChain`-like object up until the point `force` was called. Likewise, while at any point in the `pipeline` the intermediate type could change, the change was known prior to composition to ensure the proper values were fed from one stage to the next. However, what if we want to compose functions that were not necessarily meant to compose?

In this final section of this chapter, I'll talk about a technique for composing functions of incongruous return types using a new kind of lazy pipeline called `actions` (Stan 2011).

7. I'm of the opinion that code should be written for readers.

If you imagine a function as a box of indeterminate behavior taking as input data of some "shape" and outputting data of some other shape (possibly the same shape), then Figure 8-6 might be what you'd picture.[8]

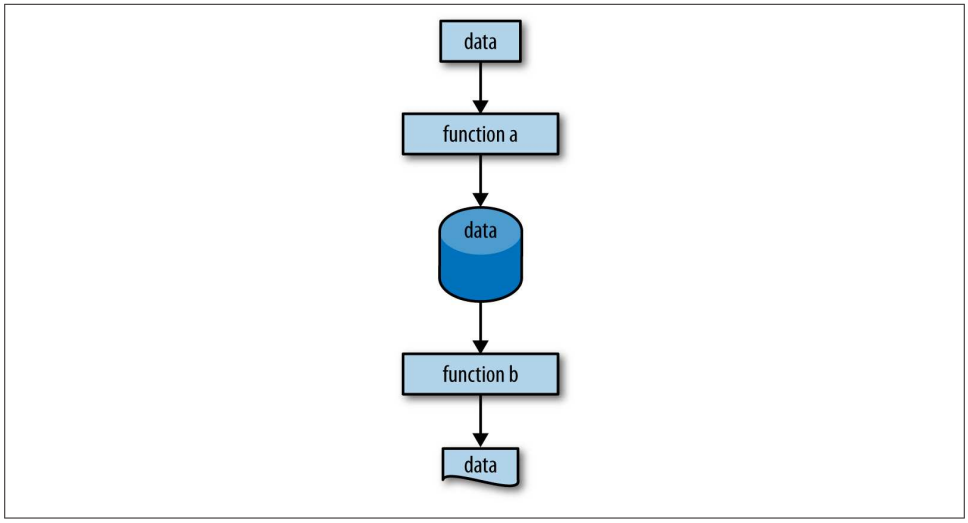


*Figure 8-6. Function a takes a rectangular shaped thing and returns a database shaped thing; function b takes a database shaped thing and returns a document shaped thing*

Therefore, the reason that a lazy chain works properly is that the shape from one chained method call to the next is consistent and only changes[9] when `force` is called. Figure 8-7 illustrates this fact.

8. By shape in this circumstance, I'm simply referring to the idea that the shape of an array of strings would be very different than the shape of a floating-point number, which in turn is different than an object of strings to arrays. You can substitute shape for "type" or "structure," if you prefer, but I'll stick to shape because the pictures look prettier. The idea of visualizing shapes was inspired by the amazing Alan Dipert (Dipert 2012).

9. Although there is no reason that the result of `force` couldn't be yet another lazy chain; but I digress.

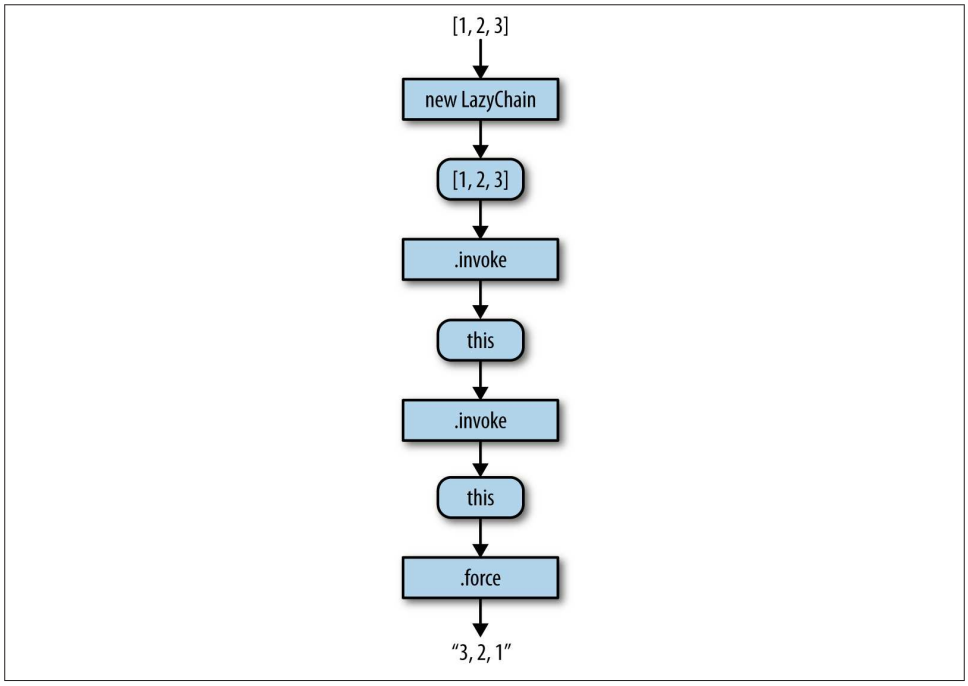


*Figure 8-7. The shape flowing between calls in a lazy chain is stable, only (potentially) changing when force is called*

Similarly, the shape between the nodes of a pipeline or a composed function, while not as stable as a common object reference, is designed to change in accordance with the needs of the next node, as shown in Figure 8-8.

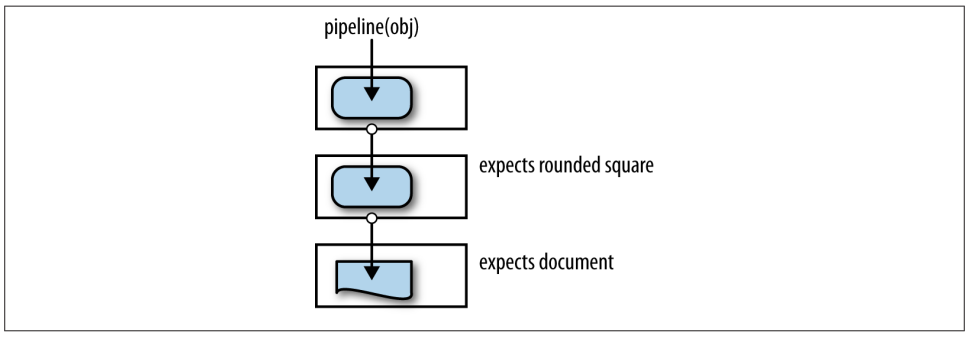


*Figure 8-8. The shape flowing between calls in a pipeline or compose is designed to change in expected ways*

The problem is that if the shapes do not align, then neither `pipeline`, `_.compose`, nor `lazyChain` will operate as expected:

```
pipeline(42
  , sqr
  , note
  , function(n) { return -n });

// NOTE: 1764
//=> NaN
```

Not cool. The reason that failed was because the shape of the type changed in midstream to `undefined` (from `note`).

In fact, if you want to achieve the correct effect, then you'd need to do so manually:

```
function negativeSqr(n) {
  var s = sqr(n);
  note(n);
  return -s;
}

negativeSqr(42);
// NOTE: 1764
//=> -1764
```

While tenable, the amount of boilerplate involved in getting this to work for larger capabilities grows quickly. Likewise, I could just change the `note` function to return whatever it's given, and while that might be a good idea in general, doing so here would solve only a symptom rather than the larger disease of incompatible intermediate shapes. That there are functions that can return incompatible shapes, or even no shape at all (i.e., no `return`) requires a delicate orchestration of control flow to compose code. The requirements of this delicate balance work against us in finding a way to compose functions that flow values from one to the next.

By now you might think that the way to fix this problem is to somehow find a way to stabilize the shapes flowing between the nodes—that thinking is absolutely correct.

## Finding a Common Shape

The complication in determining a common shape to flow between nodes of a different sort is not picking a type (a regular object will do), but what to put into it. One choice is the data that flows between each node; in the `negativeSqr` example, the object would look like the following:

```
{values: [42, 1764, undefined, -1764]}
```

But what else is needed? I would say that a useful piece of data to keep around would be the state, or target object used as the common target between nodes. Figure 8-9 shows

a way to visualize how actions could be composed, even in the face of disparate input and output shapes.

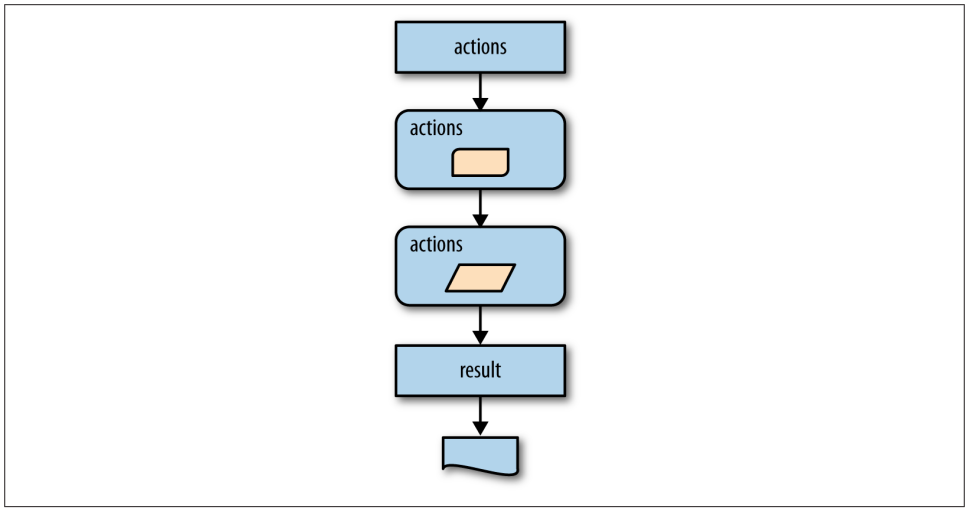


*Figure 8-9. The shape flowing between actions is made to be stable using a context object*

The last node (i.e., results) operates much in the same way as `force` in that it pulls the answer out of the action object into the real world. In the case of the `negativeSqr` function, the way to get the final answer is to retrieve the last element of the `values` element or just return the state:

```
{values: [42, 1764, undefined, -1764],
 state:  -1764}
```

Now, the implementation of the `actions` function to manage these intermediate states is a hybrid of the `pipeline` and `lazyChain` implementations, as shown here:

```
function actions(acts, done) {
  return function (seed) {
    var init = { values: [], state: seed };

    var intermediate = _.reduce(acts, function (stateObj, action) {
      var result = action(stateObj.state);
      var values = cat(stateObj.values, [result.answer]);

      return { values: values, state: result.state };
    }, init);

    var keep = _.filter(intermediate.values, existy);

    return done(keep, intermediate.state);
```

```
  };
};
```

The `actions` function expects an array of functions, each taking a value and returning a function that augments the intermediate state object. The `actions` function then reduces over all of the functions in the array and builds up an intermediate state object, as shown here:

```
...
    var intermediate = _.reduce(acts, function (stateObj, action) {
      var result = action(stateObj.state);
      var values = cat(stateObj.values, [result.answer]);

      return { values: values, state: result.state };
    }, init);
...
```

During this process, `actions` expects the result from each function to be an object of two keys: `answer` and `state`. The `answer` value corresponds to the result of calling the function and the `state` value represents what the new state looks like after the "action" is performed. For a function like `note`, the state does not change. The `intermediate` state object might have some bogus answers in it (e.g., the `answer` of `note` is `undefined`), so `actions` filters those out:

```
...
    var keep = _.filter(intermediate.values, existy);

    return done(keep, intermediate.state);
...
```

Finally, `actions` passes the filtered `values` (called `keep`) and `state` into the `done` function to garner a final result. I could have only passed the `state` or `values` into the `done` function, but I like to pass both for maximum flexibility, and because it helps for illustration.

To demonstrate how `actions` works, I'll need to break apart `negativeSqr` and recompose it as a series of actions. First, the `sqr` function obviously doesn't know anything about a state object, so I'll need to create an adapter function, called `mSqr`:[10]

```
function mSqr() {
  return function(state) {
    var ans = sqr(state);
    return {answer: ans, state: ans};
  }
}
```

10. Monad is the proper term for `action`, but I hesitate to use it in this section because I think that monads are vastly weakened in the absence of a strong-type system and return-type polymorphism. However, that's not to say that monads cannot teach us valuable lessons in deconstruction for use in JavaScript.

I can now use `actions` just to perform a double-squaring operation:

```
var doubleSquareAction = actions(
  [mSqr(),
   mSqr()],
  function(values) {
    return values;
});

doubleSquareAction(10);
//=> [100, 10000]
```

Since I returned the `values` array directly, the result of `doubleSquareAction` is all of the intermediate states (specifically the square of 10 and the square of the square of 10). However, this is almost the same as `pipeline`. The real magic comes when mixing functions of differing shapes:

```
function mNote() {
  return function(state) {
    note(state);
    return {answer: undefined, state: state};
  }
}
```

The `answer` of the `mNote` function is, of course, `undefined`, since it is a function used for printing; however, the `state` is just passed along. The `mNeg` function should by now seem apparent:

```
function mNeg() {
  return function(state) {
    return {answer: -state, state: -state};
  }
}
```

And now composing these new functions into `actions` is shown here:

```
var negativeSqrAction = actions([mSqr(), mNote(), mNeg()],
  function(_, state) {
    return state;
  });
```

Its usage is shown here:

```
negativeSqrAction(9);
// NOTE: 81
//=> -81
```

Using the `actions` paradigm for composition is a general way to compose functions of different shapes. Sadly, the preceding code seems like a lot of ceremony to achieve the effects needed. Fortunately, there is a better way to define an action, without needing to know the details of how a state object is built and avoiding the pile of boilerplate that goes along with that knowledge.

## A Function to Simplify Action Creation

In this section, I'll define a function, `lift`, that takes two functions: a function to provide the result of some action given a value, and another function to provide what the new state looks like. The `lift` function will be used to abstract away the management of the state object used as the intermediate representation of `actions`. The implementation of `lift` is quite small:

```
function lift(answerFun, stateFun) {
  return function(/* args */) {
    var args = _.toArray(arguments);

    return function(state) {
      var ans = answerFun.apply(null, construct(state, args));
      var s = stateFun ? stateFun(state) : ans;

      return {answer: ans, state: s};
    };
  };
};
```

`lift` looks like it's curried (i.e., it returns a function), and indeed it is. There is no reason to curry `lift` except to provide a nicer interface, as I'll show in a moment. In fact, using `lift`, I can more nicely redefine `mSqr`, `mNote`, and `mNeg`:

```
var mSqr2  = lift(sqr);
var mNote2 = lift(note, _.identity);
var mNeg2  = lift(function(n) { return -n });
```

In the case of `sqr` and the negation function, both the answer and the state are the same value, so I only needed to supply the answer function. In the case of `note`, however, the answer (`undefined`) is clearly not the state value, so using `_.identity` allows me to specify that it's a pass-through action.

The new actions compose via `actions`:

```
var negativeSqrAction2 = actions([mSqr2(), mNote2(), mNeg2()],
  function(_, state) {
    return state;
  });
```

And their usage is the same as before:

```
negativeSqrAction(100);
// NOTE: 10000
//=> -10000
```

If I want to use `lift` and actions to implement a `stackAction`, then I could do so as follows:

```
var push = lift(function(stack, e) { return construct(e, stack) });
```

The push function returns a new array, masquerading as a stack, with the new element at the front. Since the intermediate state is also the answer, there is no need to supply a state function. The implementation of pop needs both:

```
var pop = lift(_.first, _.rest);
```

Since I'm simulating a stack via an array, the pop answer is the first element. Conversely, the state function `_.rest` return the new stack with the top element removed. I can now use these two functions to compose two pushes and one pop, as follows:

```
var stackAction = actions([
  push(1),
  push(2),
  pop()
  ],
  function(values, state) {
    return values;
  });
```

Amazingly, by using the `actions` function, I've captured the sequence of stack events as a value that has not yet been realized. To realize the result is as simple as this:

```
stackAction([]);

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

As shown, the `stackAction` is just a function and can now be composed with other functions to build higher-level behaviors. Since I've decided to return all of the intermediate answers, the resulting return value can participate in a vast array of composition scenarios:

```
pipeline(
  []
  , stackAction
  , _.chain)
.each(function(elem) {
  console.log(polyToString(elem))
});

// (console) [[1],      // the stack after push(1)
// (console)  [2, 1],   // the stack after push(2)
// (console)  2]        // the result of pop([2, 1])
```

This is almost like magic, but by deconstructing it, I've tried to show that it really isn't magical at all. Instead, composing functions of different shapes is possible using a common intermediate type and a couple of functions—`lift` and `actions`—to manage them along the way. This management allows me to convert a problem that would typically be a problem of control flow in keeping the types straight, into a problem of data flow—the whole point of this chapter (Piponi 2010).

## Summary

This chapter focused on exploring the possibilities in viewing behavior as a sequence of discrete steps. In the first part of the chapter, I discussed chaining. Method chaining is a common technique in JavaScript libraries, reaching the widest audience in jQuery. In summary, method chaining is the act of writing object methods to all return a common `this` reference so common methods can be called in sequence. Using jQuery promises and Underscore's `_.chain` function, I explored chaining. However, I also explored the idea of "lazy chaining," or sequencing some number of method calls on a common target for later execution.

Following on the idea of a chain was that of the "pipeline," or a sequence of function calls that take in a piece of data and return a transformed piece of data at the other end. Pipelines, unlike chains, work against data such as arrays and objects rather than a common reference. Also, the type of data flowing through a pipeline can change as long as the next step in the pipeline expects that particular type. As discussed, pipelines are meant to be pure—no data was harmed by running it through.

While both chains and pipelines work against either a known reference or data types, the idea of a sequence of actions is not limited to doing so. Instead, the implementation of the `actions` type hides the details of managing an internal data structure used to mix functions of varying return and argument types.

In the next and final chapter, I will talk about how functional programming facilitates and indeed motivates a "classless" style of programming.