CHAPTER 7

Purity, Immutability, and Policies for Change

This chapter marks the point when a fully functional and practical style is explored. Functional programming is not just about functions; it's also a way of thinking about how to build programs to minimize the complexities inherent in the creation of software. One way of reducing the complexity is to reduce or eliminate (ideally) the footprint of state change taking place in our programs.

Purity

Imagine that you needed a function that, when given a number, returned a (pseudo) random number greater than `0` and up to and including the number itself. Underscore's `_.random` function is almost correct, but it defaults to including zero. Therefore, as a first approximation, you might write something like this:

```
var rand = partial1(_.random, 1);
```

Using `rand` is as simple as the following:

```
rand(10);
//=> 7

repeatedly(10, partial1(rand, 10));
//=> [2, 6, 6, 7, 7, 4, 4, 10, 8, 5]

_.take(repeatedly(100, partial1(rand, 10)), 5);
//=> [9, 6, 6, 4, 6]
```

You can use `rand` as the basis for a generator for random lowercase ASCII strings-with-numbers of a certain length as follows:

```
function randString(len) {
  var ascii = repeatedly(len,  partial1(rand, 26));
```

```
  return _.map(ascii, function(n) {
    return n.toString(36);
  }).join('');
}
```

And here's the use of `randString`:

```
randString(0);
//=> ""

randString(1);
//=> "f"

randString(10);
//=> "k52k7bae8p"
```

Building the `randString` function is just like what I've shown throughout the course of this book. Plugging functions into functions to build higher-level capabilities has been what I've been building up to all this time, right? While `randString` technically fits this definition, there is one big difference in the way that `randString` is constructed from the way that the previous functions were. Can you see it? The answer lies in another question. Can you test it?

The Relationship Between Purity and Testing

How would you test the function `randString`? That is, if you were using something like Jasmine[1] to build a spec for the `randString` function, how would you complete the following code fragment?

```
describe("randString", function() {
  it("builds a string of lowercase ASCII letters/digits", function() {
    expect(randString()).to???(???);
  });
});
```

What validation function and value could you put into the parts labeled `???` to make the specification pass? You could try to add a given string, but that would be a waste of time, given that the whole point is to generate randomness. It may start to be clear now that the problem with `randString` is that there is no way to predict what the result of a call will be. This condition is very different from a function like `_.map`, where every call is determinable from the arguments presented to it:

```
describe("_.map", function() {
  it("should return an array made from...", function(){
    expect(.map([1,2,3], sqr)).toEqual([1, 4, 9]);
  });
```

1. Jasmine is a lovely test framework that I personally use and highly recommend.

```
});
{
    expect(_.map([1,2,3], sqr)).toEqual([1, 4, 9]);
  });
});
```

The operation of `_.map` as just described is know as "pure." A pure function adheres to the following properties:

- Its result is calculated only from the values of its arguments.
- It cannot rely on data that changes external to its control.
- It cannot change the state of something external to its body.

In the case of `randString`, the first rule of purity is violated because it doesn't take any arguments to use in a calculation. The second rule is violated because its result is entirely based on JavaScript's random number generator, which is a black-box taking no input arguments and producing opaque values. This particular problem is a problem at the language level and not at the level of generating randomness. That is, you could create a random number generator that was pure by allowing the caller to supply a "seed" value.

Another example of a function that breaks rule #1 is as follows:

```
PI = 3.14;

function areaOfACircle(radius) {
  return PI * sqr(radius);
}

areaOfACircle(3);
//=> 28.26
```

You probably already see where the problem lies, but for the sake of completeness, assume that within a web page, another library is loaded with the following code fragment:

```
// ... some code

PI = "Magnum";

// ... more code
```

What is the result of calling `areaOfACircle`? Observe:

```
areaOfACircle(3);
//=> NaN
```

Whoops!

This kind of problem is especially nasty in JavaScript because of its ability to load arbitrary code at runtime that can easily change objects and variables. Therefore, to write functions that rely on data outside of its control is a recipe for confusion. Typically, when

you attempt to test functions that rely on the vagaries of external conditions, all test cases *must* set up those same conditions for the very purpose of testing. Observing a functional style that adheres to a standard of purity wherever possible will not only help to make your programs easier to test, but also easier to reason about in general.

Separating the Pure from the Impure

Because JavaScript's `Math.rand` method is impure by design, any function that uses it is likewise impure and likely more difficult to test. Pure functions are tested by building a table of input values and output expectations. Other methods and functions within JavaScript that infect code with impurity are `Date.now`, `console.log`, `this`, and use of global variables (this is not a comprehensive list). In fact, because JavaScript passes object references around, every function that takes an object or array is potentially subject to impurity. I'll talk later in this section about how to alleviate these kinds of problems, but the gist of this is that while JavaScript can never be completely pure (nor would we want that), the effects of change can be minimized.

While the `randString` function is undoubtedly impure as written, there are ways to restructure the code to separate the pure from the impure parts. In the case of `randString`, the delineation is fairly clear: there is a character generation part, and a part that joins the characters together. To separate the pure from the impure, then, is as simple as creating two functions:

```
function generateRandomCharacter() {
  return rand(26).toString(36);
}

function generateString(charGen, len) {
  return repeatedly(len, charGen).join('');
}
```

Changing the implementation to `generateString` (which explicitly takes a function intended for character generation) allows the following patterns of usage:

```
generateString(generateRandomCharacter, 20);
//=> "2lfhjo45n2nfnpbf7m7e"
```

Additionally, because `generateString` is a higher-order function, I can use `partial` to compose the original, impure version of `randomString`:

```
var composedRandomString = partial1(generateString, generateRandomCharacter);

composedRandomString(10);
//=> "j18obij1jc"
```

Now that the pure part is encapsulated within its own function, it can be tested independently:

```
describe("generateString", function() {
  var result = generateString(always("a"), 10);
```

```
  it("should return a string of a specific length", function() {
    expect(result.constructor).toBe(String);
    expect(result.length).toBe(10);
  });

  it("should return a string congruent with its char generator", function() {
    expect(result).toEqual("aaaaaaaaaa");
  });
});
```

There's still a problem testing the validity of the impure `generateRandomCharacter` function, but it's nice to have a handle on a generic, easily testable capability like `generateString`.

Property-Testing Impure Functions

If a function is impure, and its return value is subject to conditions outside of its control, then how can it be tested? Assuming that you've managed to reduce the impure part to its bare minimum, like with `generateRandomCharacter`, then the matter of testing is somewhat easier. While you cannot test the return value for specific values, you can test it for certain characteristics. In the example of `generateRandomCharacter`, I could test for the following characteristics:

- ASCII-ness
- Digit-ness
- String-ness
- Character-ness
- Lowercase-ness

To check each of these characteristics requires a lot of data, however:

```
describe("generateRandomCharacter", function() {
  var result = repeatedly(10000, generateRandomCharacter);

  it("should return only strings of length 1", function() {
    expect(_.every(result, _.isString)).toBeTruthy();
    expect(_.every(result, function(s) { return s.length === 1 })).toBeTruthy();
  });

it("should return a string of only lowercase ASCII letters or digits", function()
{
  expect(_.every(result, function(s) {
    return /[a-z0-9]/.test(s) })).toBeTruthy();


    expect(_.any(result, function(s) { return /[A-Z]/.test(s) })).toBeFalsy();
```

```
  });
});
```

Testing the characteristics of only `10000` results of calls to `generateRandomCharacter` is not enough for full test coverage. You can increase the number of iterations, but you'll never be fully satisfied. Likewise, it would be nice to know that the characters generated fall within certain bounds. In fact, there is a limitation in my implementation that restricts it from generating every possible legal lowercase ASCII character, so what have I been testing? I've been testing the incorrect solution. Solving the problem of creating the wrong thing is a philosophical affair, far outside the depth of this book. For the purposes of random password generation this might be a problem, but for the purposes of demonstrating the separation and testing of impure pieces of code, my implementation should suffice.

Purity and the Relationship to Referential Transparency

Programming with pure functions may seem incredibly limiting. JavaScript, as a highly dynamic language, allows the definition and use of functions without a strict adherence to the types of their arguments or return value. Sometimes this loose adherence proves problematic (e.g., `true + 1 === 2`), but other times you know exactly what you're doing and can take advantage of the flexibility. Very often, however, JavaScript programmers equate the ability of JavaScript to allow free-form mutation of variables, objects, and array slots as essential to dynamism. However, when you exercise a libertarian view of state mutation, you're actually limiting your possibilities in composition, complicating your ability to reason through the effects of any given statement, and making it more difficult to test your code.

Using pure functions, on the other hand, allows for the easy composition of functions and makes replacing any given function in your code with an equivalent function, or even the expected value, trivial. Take, for example, the use of the `nth` function to define a `second` function from Chapter 1:

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

The `nth` function is a pure function. That is, it will adhere to the following for any given array argument. First, it will always return the same value given some array value and index value:

```
nth(['a', 'b', 'c'], 1);
//=> 'b'

nth(['a', 'b', 'c'], 1);
// 'b'
```

You could run this call a billion times and as long as `nth` receives the array `['a', 'b', 'c']` and the number `1`, it will always return the string `'b'`, regardless of the state of

anything else in the program. Likewise, the `nth` function will never modify the array given to it:

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

nth(a, 1);
//=> 'b'

a === a;
//=> true

nth(a, 1);
//=> 'b'

_.isEqual(a, ['a', 'b', 'c']);
//=> true
```

The one limiting factor, and it's one that we've got to live with in JavaScript, is that the `nth` function might return something that's impure, such as an object, an array, or even an impure function:

```
nth([{a: 1}, {b: 2}], 0);
//=> {a: 1}

nth([function() { console.log('blah') }], 0);
//=> function ...
```

The only way to rectify this problem is to observe a strict adherence to the use and definition of pure functions that do not modify their arguments, nor depend on external values, except where such effects have been minimized explicitly. Realizing that some discipline is required to maintain functional purity, we will be rewarded with programming options. In the case of `second`, I can replace the definition of `nth` with something equivalent and not miss a beat:[2]

```
function second(a) {
  return a[1];
}
```

Or maybe:

```
function second(a) {
  return _.first(_.first(a));
}
```

In either of these cases, the behavior of `second` has not changed. Because `nth` was a pure function, its replacement in this case was trivial. In fact, because the `nth` function is

2. That's not exactly true because `nth` checks array bounds and throws an error when an index exceeds the array's length. When changing underlying implementations, be aware of the tangential effects of the change in addition to gains in raw speed.

pure, it could conceivably be replaced with the value of its result for a given array and still maintain program consistency:

```
function second() {
  return 'b';
}

second(['a', 'b', 'c'], 1);
//=> 'b'
```

The ability to freely swap new functions without the confusion brought on by the balancing act of mutation is a different way to look at freedom in program composition. A related topic to purity and referential transparency is the idea of idempotence, explained next.

Purity and the Relationship to Idempotence

With the growing prevalence of APIs and architectures following a RESTful style, the idea of idempotence has recently thrust itself into the common consciousness. Idempotence is the idea that executing an activity numerous times has the same effect as executing it once. Idempotence in functional programming is related to purity, but different enough to bear mention. Formally, a function that is idempotent should make the following condition true:

```
someFun(arg) == _.compose(someFun, someFun)(arg);
```

In other words, running a function with some argument should be the same as running that same function twice in a row with the same argument as in `someFun(someFun(arg))`. Looking back on the `second` function, you can probably guess that it's not idempotent:

```
var a = [1, [10, 20, 30], 3];

var secondTwice = _.compose(second, second);

second(a) === secondTwice(a);
//=> false
```

The problem, of course, is that the bare call to `second` returns the array `[10, 20, 30]`, and the call to `secondTwice` returns the nested value `20`. The most straightforward idempotent function is probably Underscore's `_.identity` function:

```
var dissociativeIdentity = _.compose(_.identity, _.identity);

_.identity(42) === dissociativeIdentity(42);
//=> true
```

JavaScript's `Math.abs` method is also idempotent:

```
Math.abs(Math.abs(-42));
//=> 42
```

You need not sacrifice dynamism by adhering to a policy of pure functions. However, bear in mind that any time that you explicitly change a variable, be it encapsulated in a closure, directly or even in a container object (later this chapter), you introduce a time-sensitive state. That is, at any given tick of the program execution, the total state of the program is dependent on the subtle change interactions occurring. While you may not be able to eliminate all state change in your programs, it's a good idea to reduce it as much as possible. I'll get into isolated change later in this chapter, but first, related to functional purity is the idea of immutability, or the lack of explicit state change, which I'll cover next.

Immutability

Very few data types in JavaScript are immutable by default. Strings are one example of a data type that cannot be changed:

```
var s = "Lemongrab";

s.toUpperCase();
//=> "LEMONGRAB"

s;
//=> "Lemongrab"
```

It's a good thing that strings are immutable because scenarios like the following might occur, wreaking mass confusion:[3]

```
var key = "lemongrab";
var obj = {lemongrab: "Earl"};

obj[key] === "Earl";
//=> true

doSomethingThatMutatesStrings(key);

obj[key];
//=> undefined

obj["lemonjon"];
//=> "Earl"
```

This would be an unfortunate sequence of events. You'd likely find the problem with some digging, but if there was a widespread culture of string mutating, then these kinds of problems would pop up far more frequently than you'd like. Thankfully, that strings

3. The Ruby programming language allows string mutation, and prior to version 1.9 fell victim to this kind of trap. However, Ruby 1.9 Hash objects copy string keys and are therefore shielded. Unfortunately, it still allows mutable objects as keys, so mutating those can and will break Hash lookups.

in JavaScript are immutable eliminates a whole class of nasty problems. However, the following mutation *is* allowed in JavaScript:[4]

```
var obj = {lemongrab: "Earl"};

(function(o) {
  _.extend(o, {lemongrab: "King"});
})(obj);

obj;
//=> {lemongrab: "King"}
```

While we're happy that strings are immutable, we tend not to blink an eye over the fact that JavaScript objects are mutable. In fact, much of JavaScript has been built to take advantage of mutability. However, as JavaScript gains more acceptability in industry, larger and larger programs will be written using it. Imagine a depiction of the dependencies created by points of mutation within a very small program as shown in Figure 7-1.

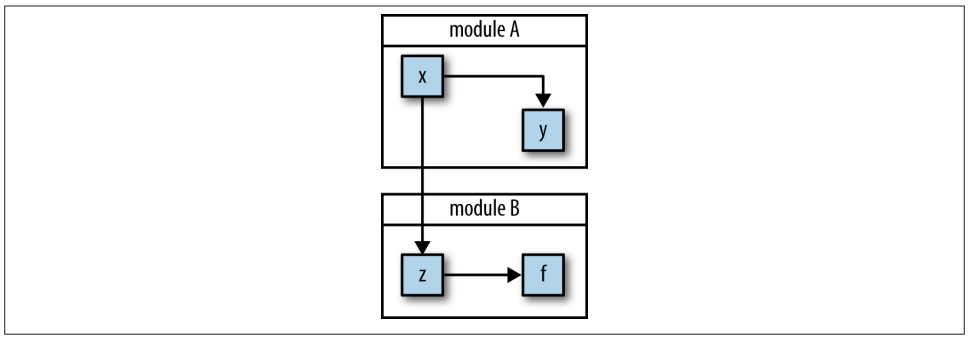

Figure 7-1. Even in small programs, the "web of mutation" is tangled, but it may be manageable

However, as the program grows, the "web of mutation" likewise grows, gaining more and more edges from one change dependency to the next, as shown in Figure 7-2.

4. Underscore's `extend` function fooled me once, but really it was my own prejudices that allowed me to assume that it was a pure function. Once I learned that it was not, I realized a fun way to take advantage of that fact, as you'll see in Chapter 9.

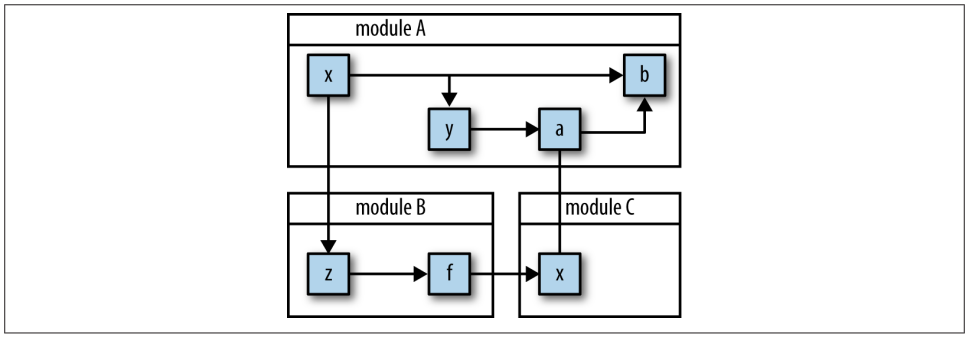

Figure 7-2. As programs grow, so grows the "web of mutation"

This state of affairs cannot be easily maintained. If every change affecting the web of mutation causes widespread disturbances in the delicate relationships between the states and their dependents, then *any* change affects the whole.[5] In functional programming, the ideal situation is that there is *never* mutation, and if you start with a policy of immutability, you'd be surprised how far you can get. In this section, I'll discuss the virtues of immutability and how to observe its dictum.

If a Tree Falls in the Woods, Does It Make a Sound?

Throughout this book, you'll notice that I've often used mutable arrays and objects within the implementations of many functions. To illustrate what I mean, observe the implementation of a function, `skipTake`, that when given a number `n` and an array, returns an array containing every nth element:

```
function skipTake(n, coll) {
  var ret = [];
  var sz = _.size(coll);

  for(var index = 0; index < sz; index += n) {
    ret.push(coll[index]);
  }

  return ret;
}
```

The use of `skipTake` is as follows:

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

5. That's not to say that all JavaScript programs work that way. In the past few years, there has been a growing focus on discipline in design. One article of particular note along this theme is "Don't Modify Objects You Don't Own" (Zakas 2010).

```
skipTake(3, _.range(20));
//=> [0, 3, 6, 9, 12, 15, 18]
```

Within the implementation of `skipTake`, I very deliberately used an array coupled with an imperative loop performing an `Array#push`. There are ways to implement `skipTake` using functional techniques, therefore requiring no explicit mutation. However, the `for` loop implementation is small, straightforward, and fast. More importantly, the use of this imperative approach is completely hidden from the users of the `skipTake` function. The advantage of viewing the function as the basic unit of abstraction is that within the confines of any given function, implementation details are irrelevant as long as they do not "leak out." By "leak out" I mean that you can use a function as a boundary for local state mutation, shielding change from the sight of external code.

Whether I used `_.foldRight` or `while` within `skipTake` is irrelevant to the users of the function. All that they know, or care about, is that they will get a new array in return and that the array that they passed in will not be molested.

> If a tree falls in the woods, does it make a sound?
>
> If a pure function mutates some local data in order to produce an immutable return value, is that OK?
>
> —Rich Hickey at *http://clojure.org/transients*

As it turns out, the answer is yes.[6]

Immutability and the Relationship to Recursion

If you've read as many books on functional programming as me (or even two), then an interesting pattern emerges. In almost every case, the topic of recursion and recursive techniques is covered. There are many reasons why this is the case, but one important reason relates to purity. In many functional programming languages, you cannot write a function like `summ` using local mutation:

```
function summ(array) {
  var result = 0;
  var sz = array.length;

  for (var i = 0; i < sz; i++)
    result += array[i];

  return result;
```

6. The function is a convenient boundary for hiding mutation, but its not the only one. As I'll show in Chapter 8, there are larger boundaries available to hide mutation. Historically, objects have served as nice boundaries to hide mutations, and even whole libraries and systems have been written to leverage the inherent speed of mutation while still presenting a nicely functional public facade.

```
}

summ(_.range(1,11));
//=> 55
```

The problem is that the function `summ` mutates two local variables: `i` and `result`. However, in traditional functional languages, local variables are not actually variables at all, but are instead immutable and cannot change. The only way to modify the value of a local is to change it via the call stack, and this is exactly what recursion does. Below is a recursive implementation of the same function:

```
function summRec(array, seed) {
  if (_.isEmpty(array))
    return seed;
  else
    return summRec(_.rest(array), _.first(array) + seed);
}

summRec([], 0);
//=> 0

summRec(_.range(1,11), 0);
//=> 55
```

When using recursion, state is managed via the function arguments, and change is modeled via the arguments from one recursive call to the next.[7] JavaScript allows this kind of recursive state management, with recursion depth limits as mentioned in Chapter 6, but it also allows for the mutation of local variables. So why not use the one that's faster all the time? As I'll discuss in the next section, there are also caveats to mutating local state.

Defensive Freezing and Cloning

Because JavaScript passes arrays and objects by reference, nothing is truly immutable. Likewise, because JavaScript object fields are always visible, there is no easy way to make them immutable (Goetz 2005). There are ways to hide data using encapsulation to avoid accidental change, but at the topmost level, all JavaScript objects are mutable, unless they are frozen.

Recent versions of JavaScript provide a method, `Object#freeze`, that when given an object or array, will cause all subsequent mutations to fail. In the case where strict mode is used, the failure will throw a `TypeError`; otherwise, any mutations will silently fail. The `freeze` method works as follows:

7. In other words, the state change in a recursive function is modeled in the stack much like I used a stack to change dynamic values way back in Chapter 3.

```
var a = [1, 2, 3];

a[1] = 42;

a;
//=> [1, 42, 3]

Object.freeze(a);
```

A normal array is mutable by default, but after the call to `Object#freeze`, the following occurs:

```
a[1] = 108;

a;
//=> [1, 42, 3]
```

That is, mutations will no longer take effect. You can also use the `Object#isFrozen` method to check if `a` is indeed frozen:

```
Object.isFrozen(a);
//=> true
```

There are two problems with using `Object#freeze` to ensure immutability:

- Unless you have complete control over the codebase, it might cause subtle (and not so subtle) errors to occur.
- The `Object#freeze` method is shallow.

Regarding the willy-nilly freezing of objects, while it might be a good idea to practice pervasive immutability, not all libraries will agree. Therefore, freezing objects and passing them around to random APIs might cause trouble. However, the deeper (ha!) problem is that `Object#freeze` is a shallow operation. That is, a freeze will only happen at the topmost level and will not traverse nested objects. Observe:

```
var x = [{a: [1, 2, 3], b: 42}, {c: {d: []}}];

Object.freeze(x);

x[0] = "";

x;
//=> [{a: [1, 2, 3], b: 42}, {c: {d: []}}];
```

As shown, attempting to mutate the array `a` fails to make a modification. However, mutating within `a`'s nested structures indeed makes a change:

```
x[1]['c']['d'] = 100000;

x;
//=> [{a: [1, 2, 3], b: 42}, {c: {d: 100000}}];
```

To perform a deep freeze on an object, I'll need to use recursion to walk the data structure, much like `deepClone` in Chapter 6:

```
function deepFreeze(obj) {
  if (!Object.isFrozen(obj))
    Object.freeze(obj);

  for (var key in obj) {
    if (!obj.hasOwnProperty(key) || !_.isObject(obj[key]))
      continue;

    deepFreeze(obj[key]);
  }
}
```

The `deepFreeze` function then does what you might expect:

```
var x = [{a: [1, 2, 3], b: 42}, {c: {d: []}}];

deepFreeze(x);

x[0] = null;

x;
//=> [{a: [1, 2, 3], b: 42}, {c: {d: []}}];

x[1]['c']['d'] = 42;

x;
//=> [{a: [1, 2, 3], b: 42}, {c: {d: []}}];
```

However, as I mentioned before, freezing arbitrary objects might introduce subtle bugs when interacting with third-party APIs. Your options are therefore limited to the following:

- Use `_.clone` if you know that a shallow copy is appropriate
- Use `deepClone` to make copies of structures
- Build your code on pure functions

Throughout this book, I've chosen the third option, but as you'll see in Chapter 8, I'll need to resort to using `deepClone` to ensure functional purity. For now, let's explore the idea of preserving immutability for the sake of purity in functional and object-centric APIs.

Observing Immutability at the Function Level

With some discipline and adherence to the following techniques, you can create immutable objects and pure functions.

Many of the functions implemented in this book, and indeed in Underscore, share a common characteristic: they take some collection and build another collection from it. Consider, for example, a function, `freq`, that takes an array of numbers or strings and returns an object of its elements keyed to the number of times they occur, implemented here:

```
var freq = curry2(_.countBy)(_.identity);
```

Because I know that the function `_.countBy` is a nondestructive operation (i.e., doesn't mutate the input array), then the composition of it and `_.identity` should form a pure function. Observe:

```
var a = repeatedly(1000, partial1(rand, 3));
var copy = _.clone(a);

freq(a);
//=> {1: 498, 2: 502}
```

Counting the frequencies of what is effectively a coin toss verifies that the result is almost a 50/50 split. Equally interesting is that the operation of `freq` did not harm the original array `a`:

```
_.isEqual(a, copy);
//=>true
```

Observing a policy of purity in function implementation helps to eliminate the worry of what happens when two or more functions are composed to form new behaviors. If you compose pure functions, what comes out are pure functions.

Because my implementation of `skipTake` was also pure, even though it used mutable structures internally, it too can be composed safely:

```
freq(skipTake(2, a));
//=> {1: 236, 2: 264}

_.isEqual(a, copy);
//=> true
```

Sometimes, however, there are functions that do not want to cooperate with a plan of purity and instead change the contents of objects with impunity. For example, the `_.extend` function merges some number of objects from left to right, resulting in a single object, as follows:

```
var person = {fname: "Simon"};

_.extend(person, {lname: "Petrikov"}, {age: 28}, {age: 108});
//=> {age: 108, fname: "Simon", lname: "Petrikov"}
```

The problem of course is that `_.extend` mutates the first object in its argument list:

```
person;
//=> {age: 108, fname: "Simon", lname: "Petrikov"}
```

So `_.extend` is off the list of functions useful for composition, right? Well, no. The beauty of functional programming is that with a little bit of tweaking you can create new abstractions. That is, rather than using object "extension," perhaps object "merging" would be more appropriate:

```
function merge(/*args*/) {
  return _.extend.apply(null, construct({}, arguments));
}
```

Instead of using the first argument as the target object, I instead stick a local empty object into the front of `_.extend`'s arguments and mutate that instead. The results are quite different, but probably as you'd expect:

```
var person = {fname: "Simon"};

merge(person, {lname: "Petrikov"}, {age: 28}, {age: 108})
//=> {age: 108, fname: "Simon", lname: "Petrikov"}

person;
//=> {fname: "Simon"};
```

Now the `merge` function can be composed with other pure functions perfectly safely—from hiding mutability you can achieve purity. From the caller's perspective, nothing was ever changed.

Observing Immutability in Objects

For JavaScript's built-in types and objects there is very little that you can do to foster pervasive immutability except pervasive freezing—or rabid discipline. Indeed, with your own JavaScript objects, the story of discipline becomes more compelling. To demonstrate, I'll define a fragment of a `Point` object with its constructor defined as follows:

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

I could probably resort to all kinds of closure encapsulation tricks[8] to hide the fact that `Point` instances do not have publicly accessible fields. However, I prefer a more simplistic approach to defining object constructors with the "private" fields marked in a special way (Bolin 2010).

8. I use this technique in Chapter 8 to implement `createPerson`.

As I'll soon show, an API will be provided for manipulating points that will not expose such implementation details. However, for now, I'll implement two "change" methods, `withX` and `withY`, but I'll do so in a way that adheres to a policy of immutability:[9]

```
Point.prototype = {
  withX: function(val) {
    return new Point(val, this._y);
  },
  withY: function(val) {
    return new Point(this._x, val);
  }
};
```

On `Point`'s prototype, I'm adding the two methods used as "modifiers," except in both cases nothing is modified. Instead, both `withX` and `withY` return fresh instances of `Point` with the relevant field set. Here's the `withX` method in action:

```
var p = new Point(0, 1);

p.withX(1000);
//=> {_x: 1000, _y: 1}
```

Calling `withX` in this example returns an instance of the `Point` object with the `_x` field set to `1000`, but has anything been changed? No:

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

As shown, the original `p` instance is the same old `[0,1]` point that was originally constructed. In fact, immutable objects by design should take their values at construction time and never change again afterward. Additionally, all operations on immutable objects should return new instances. This scheme alleviates the problem of mutation, and as a side effect, allows a nice chaining API for free:

```
(new Point(0, 1))
  .withX(100)
  .withY(-100);

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

So the points to take away are as follows:

- Immutable objects should get their values at construction time and never again change

9. Note that I excluded a `constructor` property, a la `Point.prototype = {constructor: Point, ...}`. While not strictly required for this example, it's probably best to adhere to a semblance of best practice in production code.

- Operations on immutable objects return fresh objects[10]

Even when observing these two rules you can run into problems. Consider, for example, the implementation of a `Queue` type that takes an array of elements at construction time and provides (partial) queuing logic to access them:[11]

```
function Queue(elems) {
  this._q = elems;
}

Queue.prototype = {
  enqueue: function(thing) {
    return new Queue(this._q + thing);
  }
};
```

As with `Point`, the `Queue` object takes its seed values at the time of construction. Additionally, `Queue` provides an `enqueue` method that is used to add the elements used as the seed to a new instance. The use of `Queue` is as follows:

```
var seed = [1, 2, 3];

var q = new Queue(seed);

q;
//=> {_q: [1, 2, 3]}
```

At the time of construction, the `q` instance receives an array of three elements as its seed data. Calling the `enqueue` method returns a new instance as you might expect:

```
var q2 = q.enqueue(108);
//=> {_q: [1, 2, 3, 108]}
```

And in fact, the value of `q` seems correct:

```
q;
//=> {_q: [1, 2, 3]}
```

However, all is not sunny in Philadelphia:

```
seed.push(10000);

q;
//=> {_q: [1, 2, 3, 1000]}
```

Whoops!

10. There are ways to create immutable objects that share elements from one instance to another to avoid copying larger structures entirely. However, this approach is outside the scope of this book.
11. Again, I intentionally excluded setting the `constructor` to avoid cluttering the example.

That's right, mutating the original `seed` changes the `Queue` instance that it seeded on construction. The problem is that I used the reference directly at the time of construction instead of creating a defensive clone. This time I'll implement a new object `SaferQueue` that will avoid this pitfall:

```
var SaferQueue = function(elems) {
  this._q = _.clone(elems);
}
```

A `deepClone` is probably not necessary because the purpose of the `Queue` instance is to provide a policy for element adding and removal rather than a data structure. However, it's still best to maintain immutability at the level of the elements set, which the new `enqueue` method does:

```
SaferQueue.prototype = {
  enqueue: function(thing) {
    return new SaferQueue(cat(this._q, [thing]));
  }
};
```

Using the immutability-safe `cat` function will eliminate a problem of sharing references between one `SaferQueue` instance and another:

```
var seed = [1,2,3];
var q = new SaferQueue(seed);

var q2 = q.enqueue(36);
//=> {_q: [1, 2, 3, 36]}

seed.push(1000);

q;
//=> {_q: [1, 2, 3]}
```

I don't want to lie and say that everything is safe. As mentioned before, the `q` instance has a public field `_q` that I could easily modify directly:

```
q._q.push(-1111);

q;
//=> {_q: [1, 2, 3, -1111]}
```

Likewise, I could easily replace the methods on `SaferQueue.prototype` to do whatever I want:

```
SaferQueue.prototype.enqueue = sqr;

q2.enqueue(42);
//=> 1764
```

Alas, JavaScript will only provide so much safety, and the burden is therefore on us to adhere to certain strictures to ensure that our programming practices are as safe as possible.[12]

Objects Are Often a Low-Level Operation

One final point before moving on to controlled mutation is that while the use of bare `new` and object methods is allowed, there are problems that could crop up because of them:

```
var q = SaferQueue([1,2,3]);

q.enqueue(32);
// TypeError: Cannot call method 'enqueue' of undefined
```

Whoops. I forgot the `new`. While there are ways to avoid this kind of problem and either allow or disallow the use of `new` to construct objects, I find those solutions more boilerplate than helpful. Instead, I prefer to use constructor functions, like the following:

```
function queue() {
  return new SaferQueue(_.toArray(arguments));
}
```

Therefore, whenever I need a queue I can just use the construction function:

```
var q = queue(1,2,3);
```

Further, I can use the `invoker` function to create a function to delegate to enqueue:

```
var enqueue = invoker('enqueue', SaferQueue.prototype.enqueue);

enqueue(q, 42);
//=> {_q: [1, 2, 3, 42]}
```

Using functions rather than bare method calls gives me a lot of flexibility including, but not limited to, the following:

- I do not need to worry as much about the actual types.
- I can return types appropriate for certain use cases. For example, small arrays are quite fast at modeling small maps, but as maps grow, an object may be more appropriate. This change-over can occur transparently based on programmatic use.
- If the type or methods change, then I need only to change the functions and not every point of use.
- I can add pre- and postconditions on the functions if I choose.

12. There are a growing number of JavaScript.next languages that were created because of the inconsistencies and reliance on convention with JavaScript.

- The functions are composable.

Using functions in this way is not a dismissal of object-programming (in fact, it's complementary). Instead, it pushes the fact that you're dealing with objects at all into the realm of implementation detail. This allows you and your users to work in functional abstractions and allows implementers to focus on the matter of making changes to the underlying machinery without breaking existing code.

Policies for Controlling Change

Let's be realistic. While it's wonderful if you can eliminate all unnecessary mutations and side effects in your code, there will come a time when you'll absolutely need to change some state. My goal is to help you think about ways to reduce the footprint of such changes. For example, imagine a small program represented as a dependency graph between the places where an object is created and subsequently mutated over the course of the program's lifetime.

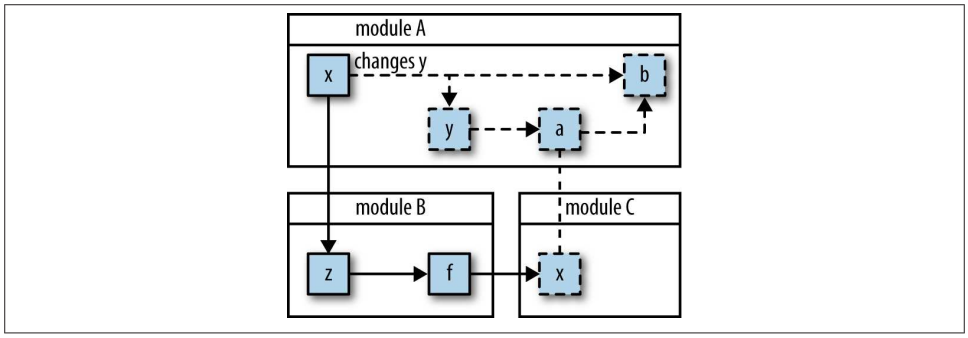

Figure 7-3. A web of mutation makes any change potentially global in its effects

Figure 7-3 should look familiar, since I talked about it earlier in this chapter. When you pass around mutable objects and modify them within one function to another method to a global scope, you've effectively lifted the effect of any change relevant to your changed object to affecting the program as a whole. What happens if you add a function that expects a certain value? What happens if you remove a method that makes a subtle mutation? What happens if you introduce concurrency via JavaScript's asynchronous operations? All of these factors work to subvert your ability to make changes in the future. However, what would it be like if change occurred only at a single point, as shown in Figure 7-4?

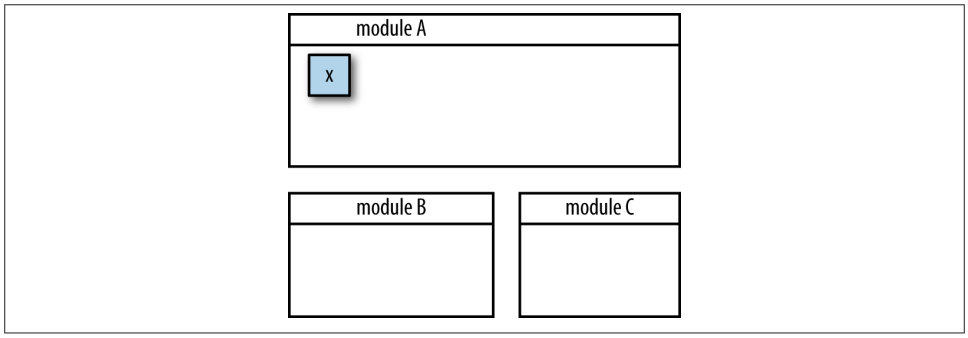

Figure 7-4. If you absolutely need to manage state, then the ideal situation is to isolate it within a single place

This is a section about isolating change to single points of mutation and strategies for achieving a compromise between the need to maintain state and to reduce complexity. The way to control the scope of change is to isolate the thing that changes. That is, rather than taking a random object and changing it in-place, a better strategy might be to hold the object in a container and change that instead. That is:

```
var container = contain({name: "Lemonjon"});

container.set({name: "Lemongrab"});
```

versus:

```
var being = {name: "Lemonjon"};

being.name = "Lemongrab";
```

While this simplistic level of indirection allows you to more easily find the place where a given value changes, it really doesn't gain much over the in-place mutation scheme. But I can take this line of thinking one step further and restrict change to occur as the result of a function call:

```
var container = contain({name: "Lemonjon"});

container.update(merge, {name: "Lemongrab"});
```

The idea behind this thinking is two-fold. First, rather than replacing the value directly, as with the fictional `container#set` method, change now occurs as the result of a function call given the current value of the container and some number of arguments. Second, by adding this functional level of indirection, change can occur based on any conceivable function, even those with domain-specific constraint checkers attached. By contrast, consider how difficult it would be to check value constraints when objects are mutated in-place, potentially at various points within your programs.

I can now show a very simple implementation of a container type:

```
function Container(init) {
  this._value = init;
};
```

Using this `Container` type is as follows:

```
var aNumber = new Container(42);

aNumber;
//=> {_value: 42}
```

However, there's more left to implement, namely the update method:

```
Container.prototype = {
  update: function(fun /*, args */) {
    var args = _.rest(arguments);
    var oldValue = this._value;

    this._value = fun.apply(this, construct(oldValue, args));

    return this._value;
  }
};
```

The thinking behind the `Container#update` method is simple: take a function and some arguments and set the new value based on a call with the existing (i.e., "old") value. Observe how this operates:

```
var aNumber = new Container(42);

aNumber.update(function(n) { return n + 1 });
//=> 43

aNumber;
//=> {_value: 43}
```

And an example that takes multiple arguments:

```
aNumber.update(function(n, x, y, z) { return n / x / y / z }, 1, 2, 3);
//=> 7.166666666666667
```

And an example showing the use of a constrained function:

```
aNumber.update(_.compose(megaCheckedSqr, always(0)));
// Error: arg must not be zero
```

This is just the beginning. In fact, in Chapter 9 I'll extend the implementation of `Container` using the idea of "protocol-based extension." However, for now, the seeds for reducing the footprint of mutation have been sown.

Summary

The chapter started by outlining and diving into functional purity, summarized as a function that does not change, return, or rely on any variable outside of its own control. While I spent a lot of time talking about functions that make no changes to their arguments, I did mention that if you need to mutate a variable internally then that was OK. As long as no one knows you've mutated a variable then does it matter? I'd say no.

The next part of the chapter talked about the related topic of immutability. Immutable data is often thwarted by JavaScript because changeable variables are the default. However, by observing certain change patterns in your program, you can get as close to immutable as possible. Again, what your callers don't know won't hurt them. Observing immutability and purity was shown to help you not only reason about your program at large, but also at the level of the unit test. If you can reason clearly about a function in isolation, then you can more easily reason about composed functions.

In the next chapter, I will cover the notion of functional "pipeline." This is very related to function composition with `_.compose`, but diving deeper into the abstraction and safety possibilities is worth devoting a chapter.