CHAPTER 4

Higher-Order Functions

This chapter builds on Chapter 3 by extending the idea that functions are first-class elements. That is, this chapter will explain that functions can not only reside in data structures and pass as data; they can return from functions, too. Discussion of these first "higher-order" functions will comprise the bulk of this chapter.

A higher-order function adheres to a very specific definition:

- It's first-class (refer back to Chapter 2 if you need a refresher on this topic)
- Takes a function as an argument
- Returns a function as a result

I've already shown many functions that take other functions as arguments, but it's worth exploring this realm more deeply, especially since its dominance is palpable in functional programming style.

Functions That Take Other Functions

You've already seen a gaggle of functions that take other functions, the more prominent being `_.map`, `_.reduce`, and `_.filter`. All of these functions adhere to the definition of higher-order. However, simply showing a few uses of each is insufficient for getting a feel for the importance of function-taking functions in functional programming. Therefore, I'll spend some time talking more about functions that take functions, and tie the practice together with a discussion of closures. Once again, whenever showing code utilizing a closure, I will capitalize the variable name of the captured value. It bears repeating that the capitalization of captured variables is not a recommended practice, but it serves well for book writing.

Thinking About Passing Functions: max, finder, and best

To start this discussion of function-taking functions, it's worth working through a few examples. Many programming languages with even modest core libraries include a function called something like `max`, which is used to find the largest value (usually a number) in a list or array. In fact, Underscore itself has such a function that performs this very task:

```
_.max([1, 2, 3, 4, 5]);
//=> 5

_.max([1, 2, 3, 4.75, 4.5])
//=> 4.75
```

There's nothing surprising in either result, but there is a limitation in this particular use case. That is, what if we want to find the maximum value in an array of objects rather than numbers? Thankfully, `_.max` is a higher-order function that takes an optional second argument. This second argument is, as you might have guessed, a function that is used to generate a numeric value from the object supplied to it.[1] For example:

```
var people = [{name: "Fred", age: 65}, {name: "Lucy", age: 36}];

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

//=> {name: "Fred", age: 65}
```

This is a very useful approach to building functions because rather than baking in the comparison of numeric values, `_.max` provides a way to compare arbitrary objects. However, this function is still somewhat limited and not truly functional. To explain, in the case of `_.max`, the comparison is always via the greater-than operator (`>`).

However, we can make a new function, `finder`, that takes two functions: one to build a comparable value, and another to compare two values and return the "best" value of the two. The implementation of `finder` is as follows:

```
function finder(valueFun, bestFun, coll) {
  return _.reduce(coll, function(best, current) {
    var bestValue = valueFun(best);
    var currentValue = valueFun(current);

    return (bestValue === bestFun(bestValue, currentValue)) ? best : current;
  });
}
```

Now, using the `finder` function, the operation of Underscore's `_.max` can be simulated via the following:

1. Underscore's `min` function works similarly.

```
finder(_.identity, Math.max, [1,2,3,4,5]);

//=> 5
```

You'll notice the use of the handy-dandy `_.identity` function that just takes a value and returns it. Seems kinda useless, right? Perhaps, but in the realm of functional programming one needs to think in terms of functions, even when the best value is a value itself.

In any case, we can now use `finder` to find different types of "best-fit" functions:

```
finder(plucker('age'), Math.max, people);

//=> {name: "Fred", age: 65}

finder(plucker('name'),
       function(x,y) { return (x.charAt(0) === "L") ? x : y },
       people);

//=> {name: "Lucy", age: 36}
```

This function of course prefers names that start with the letter `L`.

Tightening it up a bit

The implementation of `finder` is fairly small and works as we expect, but it duplicates some logic for the sake of maximum flexibility. Notice a similarity in the implementation of `finder` and the comparison logic for the best-value first-class function:

```
// in finder
return (bestValue === bestFun(bestValue, currentValue)) ? best : current);

// in the best-value function
return (x.charAt(0) === "L") ? x : y;
```

You'll notice that the logic is exactly the same in both instances. That is, both algorithms are returning some value or other based on the fitness of the first. The implementation of `finder` can be tightened by making two assumption:

- That the best-value function returns `true` if the first argument is "better" than the second argument
- That the best-value function knows how to "unwrap" its arguments

Keeping these assumptions in mind leads to the following implementation of a cleaner `best` function (Graham 1993):

```
function best(fun, coll) {
  return _.reduce(coll, function(x, y) {
    return fun(x, y) ? x : y
  });
}
```

```
best(function(x,y) { return x > y }, [1,2,3,4,5]);
//=> 5
```

With the duplication of logic removed, we now have a tighter, more elegant solution. In fact, the preceding example shows once again that the pattern `best(function(x,y) { return x > y }, ...)` provides the same functionality as Underscore's `_.max` or even `Math.max.apply(null, [1,2,3,4,5])`. Chapter 5 discusses how functional programmers capture patterns like this to create a suite of useful functions, so for now I'll defer that topic and instead hammer home the point about higher-order functions.

More Thinking About Passing Functions: repeat, repeatedly, and iterateUntil

In the previous section, I created a function, `finder`, that took two functions. As it turned out, the need to take two functions (one to unwrap a value and another to perform a comparison), was overkill for that purpose—leading to the simplification to `best`. In fact, you'll find that in JavaScript it's often overkill to create functions that take more than one function in their arguments. However, there are cases where such a creation is wholly justified, as I'll discuss in this section.

The elimination of the extra function argument to `finder` was made because the same functionality requiring two functions (i.e., unwrapping and comparison) was eliminated, due to the adoption of an assumption on the function given to `best`. However, there are circumstances where placing assumptions on an algorithm is inappropriate.

I'll walk through three related functions one by one and will discuss how they can be made more generic (and the trade-offs of doing so) along the way.

First, let me start with a very simple function, `repeat`, which takes a number and a value and builds an array containing some number of the value, duplicated:

```
function repeat(times, VALUE) {
  return _.map(_.range(times), function() { return VALUE; });
}

repeat(4, "Major");
//=> ["Major", "Major", "Major", "Major"]
```

The implementation of `repeat` uses the `_.map` function to loop over an array of the numbers `0` to `times - 1`, plopping `VALUE` into an array `4` times. You'll notice that the anonymous function closes over the `VALUE` variable, but that's not very important (nor terribly interesting, in this case) at the moment. There are many alternatives to `_.map` for implementing repeat, but I used it to highlight an important point, summarized as "use functions, not values."

Use functions, not values

The implementation of `repeat` in isolation is not a bad thing. However, as a generic implementation of "repeatness," it leaves something to be desired. That is, while a function that repeats a value some number of times is good, a function that repeats a computation some number of times is better. I can modify `repeat` slightly to perform just such a task:

```
function repeatedly(times, fun) {
  return _.map(_.range(times), fun);
}

repeatedly(3, function() {
  return Math.floor((Math.random()*10)+1);
});
//=> [1, 3, 8]
```

The function `repeatedly` is a nice illustration of the power of functional thinking. By taking a function instead of a value, I've opened up "repeatness" to a world of possibility. Instead of bottoming out immediately on a fixed value at the call-site, as `repeat` does, we can fill an array with anything. If we truly want to plug in constant values using `repeatedly`, then we need only do the following:

```
repeatedly(3, function() { return "Odelay!"; });

//=> ["Odelay!", "Odelay!", "Odelay!"]
```

In fact, the pattern illustrated by the use of a function returning a constant, no matter what its arguments will pop up various times in this book, as well as in functional libraries in the wild, but I'll talk about that more in the next section and also in Chapter 5.

You'll notice that I failed to list any parameters on the functions supplied to `repeatedly`. This was a matter of expediency, since I chose not to use the incoming arguments. In fact, because `repeatedly` is implemented as a call to `_.map` over the results of a call to `_.range`, a number representing the current repeat count is supplied to the function and could be used as you see fit. I've found this technique useful in generating some known number of DOM nodes, each with an `id` containing the repeat count value, like so:

```
repeatedly(3, function(n) {
  var id = 'id' + n;
  $('body').append($("<p>Odelay!</p>").attr('id', id));
  return id;
});

// Page now has three Odelays
//=> ["id0", "id1", "id2"]
```

In this case, I've used the jQuery library to append some nodes for me. This is a perfectly legitimate use of `repeatedly`, but it makes changes to the "world" outside of the

function. In Chapter 7 I will talk about why this is potentially problematic, but for now I'd like to make `repeatedly` even more generic.

I said, "Use functions, not values"

I've moved away from the use of the static value in `repeat`, to a function that takes one function instead in `repeatedly`. While this has indeed made `repeatedly` more open-ended, it's still not as generic as it could be. I'm still relying on a constant to determine how many times to call the given function. Often you'll know precisely how many times a function should be called, but there will be other times when knowing when to quit is not about "times" but about conditions. In other words, you may want to instead call a given function until its return value crosses some threshold, changes sign, becomes uppercase, and so on, and a simple value will not be sufficient. Instead, I can define another function that is the logical progression beyond `repeat` and `repeatedly` named `iterateUntil`; it's defined as follows:

```
function iterateUntil(fun, check, init) {
  var ret = [];
  var result = fun(init);

  while (check(result)) {
    ret.push(result);
    result = fun(result);
  }

  return ret;
};
```

The function `iterateUntil` takes two functions: a function that performs some action and another that checks a result, returning `false` if the result is a "stop" value. This is truly `repeatedly` taken to the next level in that now even the repeat count is open-ended and subject to the result of a function call. So how could you use `iterateUntil`? A simple use would be to collect all of the results of some repeated computation until the value crosses some threshold. For example, suppose you want to find all of the doubles starting at 2 up to, at most, `1024`:

```
iterateUntil(function(n) { return n+n },
             function(n) { return n <= 1024 },
             1);

//=> [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
```

To accomplish the same task with `repeatedly` requires that you know, before calling, the number of times you need to call your function to generate the correct array:

```
repeatedly(10, function(exp) { return Math.pow(2,exp+1) });

//=> [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
```

Sometimes you know how many times you need to run some calculation, and sometimes you know only how to calculate when to stop. An added advantage that `iterateUntil` provides is that the repeating loop is a feed-forward function. In other words, the result of some call to the passed function is fed into the next call of the function as its argument. I will show how this is a powerful technique later in "Pipelining" on page 176, but for now I think that we can proceed to the next section and talk about functions that return other functions.

Functions That Return Other Functions

You've already seen a few functions that return a function as a result—namely, `makeAdder`, `complement`, and `plucker`. As you might have guessed, all of these functions are higher-order functions. In this section, I will talk more in depth about higher-order functions that return (and sometimes also take) functions and closures. To start, recall my use of `repeatedly`, which used a function that ignored its arguments and instead returned a constant:

```
repeatedly(3, function() { return "Odelay!"; });

//=> ["Odelay!", "Odelay!", "Odelay!"]
```

This use of a function returning a constant is so useful that it's almost a design pattern for functional programming and is often simply called `k`. However, for the sake of clarity, I'll call it `always`; it's implemented in the following way:

```
function always(VALUE) {
  return function() {
    return VALUE;
  };
};
```

The operation of `always` is useful for illustrating some points about closures. First, a closure will capture a single value (or reference) and repeatedly return the same value:

```
var f = always(function(){});

f() === f();
//=> true
```

Because the function always produces a unique value, you can see that from one invocation of `always` to another, the captured function bound to `VALUE` is always the same (Braithwaite 2013).

Any function created with `function` will return a unique instance, regardless of the contents of its body. Using `(function(){})` is a quick way to ensure that unique values are generated. Keeping that in mind, a second important note about closures is that each new closure captures a different value than every other:

```
var g = always(function(){});

f() === g();
//=> false
```

Keeping these two rules in mind when using closures will help avoid confusion.

Moving on, plugging in `always` as a replacement for my anonymous function is a bit more succinct:

```
repeatedly(3, always("Odelay!"));

//=> ["Odelay!", "Odelay!", "Odelay!"]
```

The `always` function is what's known as a *combinator*. This book will not focus heavily on combinators, but it's worth covering the topic somewhat, as you will see them used in code bases built in a functional style. However, I will defer that discussion until Chapter 5; for now, I'd rather run through more examples of function-returning functions, especially focusing on how closures empower this approach.

However, before moving on, I'll show the implementation of another function-returning-function, `invoker`, that takes a method and returns a function that will invoke that method on any object given. Observe:

```
function invoker (NAME, METHOD) {
  return function(target /* args ... */) {
    if (!existy(target)) fail("Must provide a target");

    var targetMethod = target[NAME];
    var args = _.rest(arguments);

    return doWhen((existy(targetMethod) && METHOD === targetMethod), function() {
      return targetMethod.apply(target, args);
    });
  };
};
```

The form of `invoker` is very similar to `always`; that is, it's a function returning a function that uses an original argument, `METHOD`, during a later invocation. The returned function in this case is a closure. However, rather than returning a constant, `invoker` performs some specialized action based on the value of the original call. Using `invoker` is as follows:

```
var rev = invoker('reverse', Array.prototype.reverse);

_.map([[1,2,3]], rev);
//=> [[3,2,1]]
```

While it's perfectly legitimate to directly invoke a particular method on an instance, a functional style prefers functions taking the invocation target as an argument. Taking advantage of the fact that `invoker` returns `undefined` when an object does not have a

given method allows you to use JavaScript's natural polymorphism to build polymorphic functions. However, I'll discuss that in Chapter 5.

Capturing Arguments to Higher-Order Functions

A useful way to think about why you might create functions that return another function is that the arguments to the higher-order function serve to "configure" the behavior of the returned function. In the case of the `makeAdder` higher-order function, its argument serves to configure its returned function to always add that value to whatever number it takes. Observe:

```
var add100 = makeAdder(100);
add100(38);
//=> 138
```

Specifically, by binding the function returned by `makeAdder` to the name `add100`, I've specifically highlighted just how the return function is "configured." That is, it's configured to always add `100` to whatever you pass into it. This is a useful technique, but somewhat limited in its ability. Instead, you'll often see a function returning a function that captures a variable, and this is what I'll talk about presently.

Capturing Variables for Great Good

Imagine that you have a need for a function that generates unique strings. One such naive implementation might look like the following:[2]

```
function uniqueString(len) {
  return Math.random().toString(36).substr(2, len);
};

uniqueString(10);
//=> "3rm6ww5w0x"
```

However, what if the function needed to generate unique strings with a certain prefix? You could modify the `uniqueString` in the following way:

```
function uniqueString(prefix) {
  return [prefix, new Date().getTime()].join('');
};

uniqueString("argento");
//=> "argento1356107740868"
```

The new `uniqueString` seems to do the job. However, what if the requirements for this function change once again and it needs to return a prefixed string with an increasing

2. The primary naiveté being that there is no uniqueness guarantee on the strings generated, but I hope the intent is clear.

suffix starting at some known value? In that case, you'd like the function to behave as follows:

```
uniqueString("ghosts");
//=> "ghosts0"

uniqueString("turkey");
//=> "turkey1"
```

The new implementation can include a closure to capture some increasing value, used as the suffix:

```
function makeUniqueStringFunction(start) {
  var COUNTER = start;

  return function(prefix) {
    return [prefix, COUNTER++].join('');
  }
};

var uniqueString = makeUniqueStringFunction(0);

uniqueString("dari");
//=> "dari0"

uniqueString("dari");
//=> "dari1"
```

In the case of `makeUniqueStringFunction`, the free variable `COUNTER` is captured by the returned function and manipulated whenever it's called. This seems to work just fine, but couldn't you get the same functionality with an object? For example:

```
var generator = {
  count: 0,
  uniqueString: function(prefix) {
    return [prefix, this.count++].join('');
  }
};

generator.uniqueString("bohr");
//=> bohr0

generator.uniqueString("bohr");
//=> bohr1
```

But there is a downside to this (aside from the fact that it's not functional) in that it's a bit unsafe:

```
// reassign the count
generator.count = "gotcha";
generator.uniqueString("bohr");
//=> "bohrNaN"
```

```
// dynamically bind this
generator.uniqueString.call({count: 1337}, "bohr");
//=> "bohr1337"
```

By this time, your system is in a perilous state indeed. The approach used in `makeUniqueStringFunction` hides the instance of `COUNTER` from prying eyes. That is, the `COUNTER` variable is "private" to the closures returned. Now I'm not a stickler for private variables and object properties, but there are times when hiding a critical implementation detail from access is important. In fact, we could hide the counter in `generator` using the JavaScript secret sauce:

```
var omgenerator = (function(init) {
  var COUNTER = init;

  return {
    uniqueString: function(prefix) {
      return [prefix, COUNTER++].join('');
    }
  };
})(0);

omgenerator.uniqueString("lichking-");
//=> "lichking-0"
```

But what's the point? Creating a monstrosity like this is sometimes necessary, especially when building module/namespace-like qualifications, but it's not something that I'd like to use often.[3] The closure solution is clean, simple, and quite elegant, but it is also fraught with dread.

Take care when mutating values

I plan to talk more about the dangers of mutating (i.e., changing) variables in Chapter 7, but I can take a moment to touch on it. The implementation of `makeUniqueStringFunction` uses a little piece of state named `COUNTER` to keep track of the current value. While this piece of data is safe from outside manipulation, that it exists at all causes a bit of complexity. When a function is reliant on only its arguments for the value that it will return, it is known to exhibit something called *referential transparency*.

This seems like a fancy term, but it simply means that you should be able to replace any call to a function with its expected value without breaking your programs. When you use a closure that mutates a bit of internal code, you cannot necessarily do that because the value that it returns is wholly dependent on the number of times that it was previously called. That is, calling `uniqueString` ten times will return a different value than if it were called 10,000 times. The only way that you can replace `uniqueString` with its

3. The ECMAScript.next initiative is working through the specification of a module system that would handle visibility matters (among other things) based on simple declarations. More information is found at *http://wiki.ecmascript.org/doku.php?id=harmony:modules*.

value is if you knew *exactly* how many times it was called at any given point, but that's not possible.

Again, I will talk more about this in Chapter 7, but it's worth noting that I will avoid functions like `makeUniqueStringFunctions` unless they're absolutely necessary. Instead, I think you'll be surprised how seldom mutating a little bit of state is required in functional programming. It takes some time to change your mind-set when faced with designing functional programs for the first time, but I hope that after you finish reading this book you'll have a better idea of why a mutable state is potentially harmful, and that you will have a desire to avoid it.

A Function to Guard Against Nonexistence: fnull

Before I move into Chapter 5, I'd like to build a couple higher-order functions for illustrative purposes. The first that I'll discuss is named `fnull`. To describe the purpose of `fnull`, I'd like to show a few error conditions that it's meant to solve. Imagine that we have an array of numbers that we'd like to multiply:

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

_.reduce(nums, function(total, n) { return total * n });
//=> 0
```

Well, clearly multiplying a number by `null` is not going to give us a helpful answer. Another problem scenario is a function that takes a configuration object as input to perform some action:

```
doSomething({whoCares: 42, critical: null});
// explodes
```

In both cases, a function like `fnull` would be useful. The use for `fnull` is in a function that takes a function as an argument and a number of additional arguments, and returns a function that just calls the original function given. The magic of `fnull` is that if any of the arguments to the function that it returns are `null` or `undefined`, then the original "default" argument is used instead. The implementation of `fnull` is the most complicated higher-order function that I'll show to this point, but it's still fairly reasonable. Observe:

```
function fnull(fun /*, defaults */) {
  var defaults = _.rest(arguments);

  return function(/* args */) {
    var args = _.map(arguments, function(e, i) {
      return existy(e) ? e : defaults[i];
    });

    return fun.apply(null, args);
  };
};
```

How `fnull` works is that it circumvents the execution of some function, checks its incoming arguments for `null` or `undefined`, fills in the original defaults if either is found, and then calls the original with the patched args. One particularly interesting aspect of `fnull` is that the cost of mapping over the arguments to check for default values is incurred *only* if the guarded function is called. That is, assigning default values is done in a *lazy* fashion—only when needed.

You can use `fnull` in the following ways:

```
var safeMult = fnull(function(total, n) { return total * n }, 1, 1);

_.reduce(nums, safeMult);
//=> 30
```

Using `fnull` to create the `safeMult` function protects a product from receiving a `null` or `undefined`. This also gives the added advantage of providing a multiplication function that has an identity value when given no arguments at all.

To fix our configuration object problem, `fnull` can be used in the following way:

```
function defaults(d) {
  return function(o, k) {
    var val = fnull(_.identity, d[k]);
    return o && val(o[k]);
  };
}

function doSomething(config) {
  var lookup = defaults({critical: 108});

  return lookup(config, 'critical');
}

doSomething({critical: 9});
//=> 9

doSomething({});
//=> 108
```

This use of `fnull` ensures that for any given configuration object, the critical values are set to sensible defaults. This helps to avoid long sequences of guards at the beginning of functions and the need for the `o[k] || someDefault` pattern. Using `fnull` in the body of the `defaults` function is illustrative of the propensity in functional style to build higher-level parts from lower-level functions. Likewise, that `defaults` returns a function is useful for providing an extra layer of checks onto the raw array access.[4] Therefore,

4. The `ECMAScript.next` effort is working through a specification for default function parameters and the assignment of their values (often called optional arguments). It's unclear when this will make it into JavaScript core, but from my perspective it's a welcome feature. More information is found at *http://wiki.ecmascript.org/doku.php?id=harmony:parameter_default_values*.

using this functional style allows you to encapsulate the defaults and check logic in isolated functions, separate from the body of the `doSomething` function. Sticking with this theme, I'm going to wrap up this chapter with a function for building object-field validating functions.

Putting It All Together: Object Validators

To end this chapter, I'll work through a solution to a common need in JavaScript: the need to validate the veracity of an object based on arbitrary criteria. For example, imagine that you're creating an application that receives external commands via JSON objects. The basic form of these commands is as follows:

```
{message: "Hi!",
 type: "display"
 from: "http://localhost:8080/node/frob"}
```

It would be nice if there were a simple way to validate this message, besides simply taking it and iterating over the entries. What I would like to see is something more fluent and easily composed from parts, that reports all of the errors found with any given command object. In functional programming, the flexibility provided by functions that take and return other functions cannot be understated. In fact, the solution to the problem of command validation is a general one, with a little twist to provide nice error reporting.

Here I present a function named `checker` that takes a number of predicates (functions returning `true` or `false`) and returns a validation function. The returned validation function executes each predicate on a given object, and it adds a special error string to an array for each predicate that returns `false`. If all of the predicates return `true`, then the final return result is an empty array; otherwise, the result is a populated array of error messages. The implementation of `checker` is as follows:

```
function checker(/* validators */) {
  var validators = _.toArray(arguments);

  return function(obj) {
    return _.reduce(validators, function(errs, check) {
      if (check(obj))
        return errs
      else
        return _.chain(errs).push(check.message).value();
    }, []);
  };
}
```

The use of `_.reduce` is appropriate in this case because, as each predicate is checked, the `errs` array is either extended or left alone. Incidentally, I like to use Underscore's `_.chain` function to avoid the dreaded pattern:

```
{
  errs.push(check.message);
```

```
  return errs;
}
```

The use of `_.chain` definitely requires more characters, but it hides the array mutation nicely. (I'll talk more about hiding mutation in Chapter 7.) Notice that the `checker` function looks for a `message` field on the predicate itself. For purposes like this, I like to use special-purpose validating functions that contain their own error messages attached as pseudo-metadata. This is not a general-purpose solution, but for code under my control it's a valid use case.

A basic test for validating a command object is as follows:

```
var alwaysPasses = checker(always(true), always(true));
alwaysPasses({});
//=> []

var fails = always(false);
fails.message = "a failure in life";
var alwaysFails = checker(fails);

alwaysFails({});
//=> ["a failure in life"]
```

It's a bit of a pain to remember to set a `message` property on a validator every time you create one. Likewise, it would be nice to avoid putting properties on validators that you don't own. It's conceivable that `message` is a common enough property name that setting it could wipe a legitimate value. I could obfuscate the property key to something like `_message`, but that doesn't help the problem of remembrance. Instead, I would prefer a specific API for creating validators—one that is recognizable at a glance. My solution is a `validator` higher-order function defined as follows:

```
function validator(message, fun) {
  var f = function(/* args */) {
    return fun.apply(fun, arguments);
  };

  f['message'] = message;
  return f;
}
```

A quick check of the `validator` function bears out this strategy:

```
var gonnaFail = checker(validator("ZOMG!", always(false)));

gonnaFail(100);
//=> ["ZOMG!"]
```

I prefer to isolate the definition of individual "checkers" rather than defining them in place. This allows me to give them descriptive names, like so:

```
function aMap(obj) {
  return _.isObject(obj);
}
```

The `aMap` function can then be used as an argument to `checker` to provide a virtual sentence:

```
var checkCommand = checker(validator("must be a map", aMap));
```

And, of course, the use is as you might expect:

```
checkCommand({});
//=> true

checkCommand(42);
//=> ["must be a map"]
```

Adding straightforward checkers is just as easy. However, maintaining a high level of fluency might require a few interesting tricks. If you recall from earlier in this chapter, I mentioned that arguments to a function-returning function can serve as behavior configuration for the returned closure. Keeping this in mind will allow you to return tweaked closures anywhere that a function is expected.

Take, for example, the need to validate that the command object has values associated with certain keys. What would be the best possible way to describe this checker? I would say that a simple list of the required keys would be beautifully fluent—for example, something like `hasKeys('msg', 'type')`. To implement `hasKeys` to conform to this calling convention, return a closure and adhere to the contract of returning an error array as follows:

```
function hasKeys() {
  var KEYS = _.toArray(arguments);

  var fun = function(obj) {
    return _.every(KEYS, function(k) {
      return _.has(obj, k);
    });
  };

  fun.message = cat(["Must have values for keys:"], KEYS).join(" ");
  return fun;
}
```

You'll notice that the closure (capturing `KEYS`) does the real work of checking the validity of a given object.[5] The purpose of the function `hasKeys` is to provide an execution configuration to `fun`. Additionally, by returning a function outright, I've provided a

5. Underscore's `has` function in `hasKeys` checks an object for the existence of a keyed binding. I was tempted to use `existy(obj[k])`, but that fails when the keyed value is `null` or `undefined`, both of which are conceivably legal values.

nicely fluent interface for describing required keys. This technique of returning a function from another function—taking advantage of captured arguments along the way—is known as "currying" (I will talk more about currying in Chapter 5). Finally, before returning the closure bound to `fun`, I attach a useful `message` field with a list of all the required keys. This could be made more informative with some additional work, but it's good enough as an illustration.

Using the `hasKeys` function is as follows:

```
var checkCommand = checker(validator("must be a map", aMap),
                           hasKeys('msg', 'type'));
```

The composition of the `checkCommand` function is quite interesting. You can think of its operation as a staged validation module on an assembly line, where an argument is passed through various checkpoints and examined for validity. In fact, as you proceed through this book, you'll notice that functional programming can indeed be viewed as a way to build virtual assembly lines, where data is fed in one end of a functional "machine," transformed and (optionally) validated along the way, and finally returned at the end as *something else*.

In any case, using the new `checkCommand` checker to build a "sentence of conformity," works as you might have guessed:

```
checkCommand({msg: "blah", type: "display"});
//=> []

checkCommand(32);
//=> ["must be a map", "Must have values for keys: msg type"]

checkCommand({});
//=> ["Must have values for keys: msg type"]
```

And that nicely highlights the use of all that you've seen in this chapter. I will dig further into these topics and `checker` will make appearances again throughout this book.

Summary

In this chapter, I discussed higher-order functions that are first-class functions that also do one or both of the following:

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

To illustrate passing a function to another, numerous examples were given, including `max`, `finder`, `best`, `repeatedly`, and `iterateUntil`. Very often, passing values to functions to achieve some behavior is valuable, but sometimes such a task can be made more generic by instead passing a function.

The coverage of functions that return other functions started with the ever-valuable `always`. An interesting feature of `always` is that it returned a closure, a technique that you'll see time and time again in JavaScript. Additionally, functions returning functions allow for building powerful functions, such as `fnull` guards against unexpected `null`s, and let us define argument defaults. Likewise, higher-order functions were used to build a powerful constraint-checking system, `checker`, using very little code.

In the next chapter, I will take everything that you've learned so far and put it in the context of "composing" new functions entirely from other functions.