Lo-Dash Essentials walks you through the Lo-Dash utility library, which promises consistency and performance in JavaScript development. This book looks into the most common functions and the various contexts in which they’re used. You’ll first start with object types and their properties, then you’ll dive into larger development patterns, such as MapReduce, and how to chain functionality together. Following this, you’ll learn how to make suitable builds for various environments, and discover how high-level patterns complement one another and how they lead to reusable building blocks for applications. Finally, you will gain some practical exposure to Lo-Dash by working alongside other libraries, and learn some useful techniques for improving performance.

Who this book is written for
If you are a curious JavaScript developer interested simultaneously in tweaking the efficiency of your code, as well as improving the conciseness of it, and maintaining the readability of it, then this is the book for you. Ideally, the book is intended for readers already working on JavaScript projects and using frameworks such as jQuery and Backbone. Even if you’re already using Lo-Dash, this book will show you how to use it efficiently. While extensive JavaScript experience isn’t a requirement, you should have at least some prior programming experience in order to best understand the concepts presented.

What you will learn from this book
- Analyze the difference between arrays, collections, and objects and look at how Lo-Dash treats them
- Understand the subtle differences between iterating over arrays and objects, along with object-creation patterns used with Lo-Dash
- Exploit and work with Lo-Dash functional tools, such as partially applying arguments, and composing higher-order functions
- Explore the various approaches of MapReduce and their relationships with other Lo-Dash functions
- Build your own Lo-Dash application for various environments
- Compare Lo-Dash with other libraries and decide where Lo-Dash is a good fit
- Understand the key Lo-Dash design principles and utilize this knowledge to deliver performant user interfaces

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 "Working with Functions"
- A synopsis of the book’s content
- More information on Lo-Dash Essentials

About the Author

Adam Boduch is a senior frontend engineer from Toronto. He currently works on delivering innovative products at Virtustream. Adam loves tinkering with the latest JavaScript libraries and is passionate about high-performance user interfaces.

Having focused on writing about jQuery in jQuery UI Themes Beginner's Guide and jQuery UI Cookbook, both by Packt Publishing, Adam has ventured into a new territory with this title, Lo-Dash Essentials. He is an avid proponent of Backbone and the ecosystem that surrounds it.
Lo-Dash Essentials

At times, JavaScript can be a frustrating programming language to work with. Just when you think you have it entirely figured out, performance or cross-browser issues reveal themselves in the worst possible way. Lo-Dash is just one of many libraries that aim to help JavaScript programmers write code that's elegant, efficient, and portable. Underscore.js introduced a better way to perform functional programming with JavaScript. Lo-Dash is a continuation of this effort, and throughout this book, we'll see what sets Lo-Dash apart from other libraries out there, including Underscore.

At its heart, JavaScript is a functional language, but not necessarily a pure functional Language—you can modify variables and introduce side effects. However, functions are first-class citizens and you can do a lot with them. Lo-Dash embraces this idea and gives programmers all the tools they need to write functional code that's readable and maintainable. All the cool things you can do with higher-order functions in other languages are possible with Lo-Dash as well.

It's not just the low-level utilities and performance gains that Lo-Dash delivers, though these are always welcome; it's also the enhanced programming model and the ideas that Lo-Dash borrows from other languages such as functional and applicative programming. These are imparted to JavaScript applications as a coherent API. Underscore got the ball rolling, and Lo-Dash further enhances these ideas, albeit using different design and implementation strategies.

What This Book Covers

*Chapter 1, Working with Arrays and Collections*, explains that collections are similar to arrays but are more general than arrays. This chapter looks at the differences between these two constructs and the available Lo-Dash functions that operate on them.

*Chapter 2, Working with Objects*, teaches you how Lo-Dash treats objects as collections. This chapter looks at the available functions that work with objects.

*Chapter 3, Working with Functions*, focuses on the functional tools provided by Lo-Dash.

*Chapter 4, Transformations Using Map/Reduce*, explains that the map/reduce programming model is used often in the Lo-Dash code. This chapter offers many examples that utilize the Lo-Dash map/reduce tools in their various forms.
Chapter 5, Assembling Chains, explains how Lo-Dash can chain together functions that operate on a value. This chapter explores what chains are and how they work.

Chapter 6, Application Building Blocks, looks at the high-level application components that help organize the Lo-Dash code.

Chapter 7, Using Lo-Dash with Other Libraries, explains that Lo-Dash doesn't do everything. This chapter uses examples to illustrate where other libraries can help Lo-Dash and vice versa.

Chapter 8, Internal Design and Performance, looks at some of the internal design decisions of Lo-Dash and provides some advice on how to improve the performance of your Lo-Dash code.
Working with Functions

You'll find functions everywhere within a sufficiently large piece of JavaScript code. That's because they're treated in the same way as any other primitive type. Everything is an object in JavaScript, including functions. Functions have a context and a prototype, and they can be assigned to a new context and to variables.

Lo-Dash helps to best utilize functions. Where there are missing pieces, the utilities that Lo-Dash provides let us write some truly elegant, functional code. This chapter dives into these utilities. Whether we're changing the meaning of this or decorating an existing function, we'll walk through examples that illustrate how to get started.

In this chapter, we will cover the following topics:

- Binding function contexts
- Decorating functions
- Function constraints
- Timed execution
- Composing and currying functions

**Binding function contexts**

Every JavaScript function has a context. If you're coming from an object-oriented language, the function context is a lot like the object a method belongs to. The difference of course is that JavaScript doesn't classify objects in the object-oriented sense of the concept. Instead, functions are bound to a default context, and this can easily be changed at runtime. There are even built-in language mechanisms to make this happen.
Lo-Dash makes changing function contexts easy. We'll need to work with function contexts often when programming with Lo-Dash. We'll take a look at a number of approaches to working with and changing the context of functions now.

### Changing the this keyword

Inside a function, the execution context is referred to by the `this` keyword. This is a special binding that we don't need to declare. It's always available to reference within a given function scope. It's important to keep in mind that it's entirely up to the caller, should he/she decide to override the meaning of `this`.

The `bind()` function is a powerful way to construct a new function that is permanently bound to the specified context. Here's a first look at how `bind()` works:

```javascript
function sayWhat() {
    return 'Say, ' + this.what;
}

var sayHello = _.bind(sayWhat, {
    what: 'hello'
});

var sayGoodbye = _.bind(sayWhat, {
    what: 'goodbye'
});

sayHello();
// → "Say, hello"

sayGoodbye();
// → "Say, goodbye"
```

The preceding code defines a generic `sayWhat()` function that formats a string message based on the context for the function. In particular, it looks for the `what` property of the context. Next we use `bind()` to define two new functions based on `sayWhat()`. The `sayHello()` function is bound to a new context, while the `sayGoodbye()` function is bound to yet another context. The second argument to `bind()` is the object that becomes `this` in the function that's being bound. We can see that each of these contexts defines a unique `what` property value and this is reflected in the output of calling these two functions.

Without Lo-Dash, we would rely on the `call()`, `apply()`, or `bind()` methods of the function to change its context. The advantage with the Lo-Dash `bind()` implementation is that it performs better because it's able to optimize better than the native methods.
The `sayWhat()` function didn't make use of any arguments. But just because we're fiddling with contexts doesn't mean the function we're binding can't accept arguments. Many functions make use of both arguments passed by the caller and the context object. Functions with custom contexts can indeed accept arguments. They can also be called with additional arguments after being bound to a new context, as shown in the following code:

```javascript
function sayWhat(what) {
  if (_.isUndefined(what)) {
    what = this.what;
  }
  return 'Say, ' + what;
}

var sayHello = _.bind(sayWhat, {
  what: 'hello'
});

var sayGoodbye = _.bind(sayWhat, {}, 'goodbye'),
  saySomething = _.bind(sayWhat, {});

sayHello();
// → "Say, hello"

sayGoodbye();
// → "Say, goodbye"

saySomething('what?');
// → "Say, what?"
```

The `sayWhat()` function accepts a `what` parameter used to construct the string message. If this parameter is not supplied, it falls back to the `what` property of the context. Now we define three new functions, all with unique context and argument constraints. The `sayHello()` function isn't any different from the previous example; the `what` value is in the context. The `sayGoodbye()` function definition passes a third argument to `bind()`. After the context object, `bind()` will accept any number of arguments that are also bound to the function, but in a different way. This is called partial application, and we'll look at this later on in the chapter. The function is always bound, not only to the context, but to the argument values as well. Lastly, the `saySomething()` function is bound to a context that lacks the `what` property. Also, it is not bound to any `what` parameter. However, the `what` argument can still be supplied when the function is called, as is the case here.
Binding methods

There are no methods, per se, in JavaScript—just functions and context. However, that doesn't prevent programmers from following a more traditional object-oriented model.

If we assign a function to an object property, that object then becomes the context for the function. This is just the default behavior, and as the previous section illustrated, the context can change. However, the object to which the function belongs, being the default context, maps well to methods and encapsulation. The bindAll() function can help enforce this mapping:

```javascript
function bindName(name) {
    return _.bind(name, {
        first: 'Becky',
        last: 'Rice'
    });
}

var object = {
    first: 'Ralph',
    last: 'Crawford',
    name: function() {
        return this.first + ' ' + this.last;
    }
};

var name = bindName(object.name);

object.name();  // → "Ralph Crawford"

name();  // → "Becky Rice"

_.bindAll(object);

name = bindName(object.name)

name();  // → "Ralph Crawford"
```
Let's walk through the bits of this experiment. The goal is to illustrate that once the `bindAll()` function is applied to an object, all methods belonging to that object have the context glued to it. It cannot change after this. First, the `bindName()` function just takes another function and binds it to the `Becky` context. We'll use this later on to prove a point.

The `object` variable holds a plain object with some simple properties and a simple method. The `name` variable is a function defined using the `bindName()` function. Notice that we're taking the `object.name()` method and assigning it a new context. The values we put in the `result` object confirm this. Next is the call to `bindAll()` on `object`. From this point onward, the `name()` method context can't change—it's glued to `object`. We then proceed to prove this fact by trying to bind it to a new context again, but `bindAll()` has enforced the context.

When using `bindAll()`, you can unintentionally break other functionality in your application. The ability to change function context is a strength, not a weakness. Use `bindAll()` when you're absolutely certain that the method context should never change. If there's little to no chance of your method context changing when it shouldn't, don't bother with `bindAll()`.

The name `bindAll()` implies that this is an all or nothing operation, which actually isn't the case. We don't have to enforce the context of every method attached to your object. We can actually specify the method names as a second argument and only these methods are glued to the object context. This is illustrated in the following example:

```javascript
function getName() {
  return this.name;
}

var object = {
  name: 'My Bound Object',
  method1: getName,
  method2: getName,
  method3: getName
};

_.bindAll(object, [ 'method1', 'method2' ]);;

var method3 = _.bind(object.method3, {
```
Working with Functions

```javascript
name: 'New Context',
});

object.method1();
// → "My Bound Object"

object.method2();
// → "My Bound Object"

method3();
// → "New Context"
```

We can see that the call to `bindAll()` specifies that only `method1` and `method2` are bound to `object`. Later on, we actually try binding `method3` to a completely new context and it works as expected. Had we not limited the `bindAll()` call to specific methods, we wouldn't have been able to change the context of `method3`.

**Dynamic methods**

Methods can also lazily bind to objects. We can use the `bindKey()` function to construct a new function that will call the given method name on the given object. The method doesn't actually have to exist prior to calling `bindKey()`. That's the lazy part. And this comes in handy if you need to assign a method as a callback but aren't exactly sure if the method exists yet. Consider the following example:

```javascript
function workLeft() {
    return 65 - this.age + ' years';
}

var object = {
    age: 38
};

var work = _.bindKey(object, 'work');

object.work = workLeft;

work();
// → "27 years"
```
Here we have an object with an age property. We also have a workLeft() function that computes a number based on the age property of the context. We could assign this function directly to the work property, but we've instead used the bindKey() function to construct a new function that will reference the work() method when called. The crucial thing to note is that we're able to build this callback function before the work() method exists in object. It gets added later. It could also get swapped out for a different implementation and would still call the appropriate method.

The bound key has to exist when the function created by bindKey() is eventually called. Otherwise, you'll get a TypeError.

Just like a function that has been bound to a context using the bind() function, we still have freedom with the way arguments are managed. That is, we can bind argument values or supply argument values when the bound function is called, as shown in the following code:

```javascript
function workLeft(retirement, period) {
    return retirement - this.age + ' ' + period;
}

var collection = [
    { age: 34, retirement: 60 },
    { age: 47 },
    { age: 28, retirement: 55 },
    { age: 41 }
];

var functions = [],
    result = [];

_.forEach(collection, function(item) {
    functions.push(_.bindKey(item, 'work', item.retirement ? item.retirement : 65));
});

_.forEach(collection, function(item) {

});
```
Working with Functions

```javascript
_.extend(item, { work: workLeft });
});

_.forEach(functions, function(item) {
  result.push(item('years'));
});
// →
// [ "26 years", "18 years", "27 years", "24 years" ]
```

The `workLeft()` function depends on a couple of arguments and the `age` property of the context. Next, we define a collection of objects and a couple of empty arrays to perform our experiment. Now we have three `forEach()` iterations that demonstrate how arguments work with `bindKey()`.

The first iteration is over the collection and is where the `bindKey()` function is applied in order to generate a `work()` method. We can see that not every object in the collection has a `retirement` property value. If it doesn't, we bind 65 as the argument value. At this point, we have an array of functions, each bound to the `work()` method of their object. The second iteration populates the `work` property of each object in the collection, so now `work()` is a callable function.

The last iteration calls each of these bound method functions with another argument.

Decorating functions

A decorator does what the name implies. It decorates functions with additional capabilities. It's like an adornment for a piece of functionality. For example, let's say we've already implemented a function that looks up data in some structure. It's already used throughout our application, but now we're implementing a new component that requires this same functionality and something extra. We can use the function-decorating tools provided by Lo-Dash to take existing functions and extend them.

There are two flavors of Lo-Dash function decoration: **Partials**, which construct new functions that have the arguments of the original function partially supplied, and **Wrappers**, which build a new function that wraps the original function with a whole new function.
Partials

To create a partial function using Lo-Dash, you use the partial() function. The resulting function then has some arguments presupplied—we don't have to supply them again when called. This concept is really useful when we need to dynamically supply arguments to a function, just before it's passed to a new context where those arguments aren't available. This is also the case with callbacks, as shown in the following example:

```javascript
function sayWhat(what) {
    return 'Say, ' + what;
}

var hello = _.partial(sayWhat, 'hello'),
    goodbye = _.partial(sayWhat, 'goodbye');

hello();
    // → "Say, hello"

goodbye();
    // → "Say, goodbye"
```

The sayWhat() function builds a simple string based on the supplied string argument. The two calls to partial() that follow supply this argument. The hello() and goodbye() functions, when called, will call sayWhat() with their respective partial arguments.

As we've seen so far in this chapter, many of the Lo-Dash functions that deal with functions return new ones. They also support the arguments passed by the caller. This is valuable because adding new parameters to our functions doesn't require changes to our function bindings, as shown here:

```javascript
function greet(greeting, name) {
    return greeting + ', ' + name;
}

var hello = _.partial(greet, 'hello'),
    goodbye = _.partial(greet, 'goodbye');

hello('Fran');
    // → "hello, Fran"

goodbye('Jacob');
    // → "goodbye, Jacob"
```
The `greet()` function in the preceding code accepts two arguments, `greeting` and `name`. The `hello()` and the `goodbye()` functions are constructed as partial functions that call `greet()` with the first argument already supplied. Later on, when these functions are called, we can supply the more context-specific argument—`name`.

What if the context-specific argument were the first function argument? Can we still have the caller of the partial function supply the name? To answer this question, we turn to the `partialRight()` function:

```javascript
function greet(name, greeting) {
    return greeting + ', ' + name;
}

var hello = _.partialRight(greet, 'hello'),
    goodbye = _.partialRight(greet, 'goodbye');

hello('Brent');
// → "hello, Brent"

goodbye('Alison');
// → "goodbye, Alison"
```

This code looks similar to the previous example, but there is one important difference. The `greet()` function expects the `name` parameter as the first argument. We want the caller to be able to specify this value, but we also want to specify `greeting` as a partial argument. The `partialRight()` function works the same as `partial()` except that it passes arguments to the function in a different order.

Partials aren't limited to our own functions. We can exploit this shorthand against Lo-Dash functionality itself. If you need to run a Lo-Dash function, in a callback for example, you can construct a new partial function that redefines the Lo-Dash function, with the arguments presupplied. This is shown in the following code:

```javascript
var collection = [
    'Sheila',
    'Kurt',
    'Wade',
    'Kyle'
];

var random = _.partial(_.random, 1, collection.length),
    sample = _.partial(_.sample, collection);

random();
```
Here we have a simple collection and two partial functions that operate on it. First, we utilize the random() Lo-Dash function, supplying the range as partial arguments. Then we utilize the sample() function, supplying the collection to sample as a partial argument.

Function decorators
We can utilize the wrap() function to decorate a value or another function with specific behavior. As with all other Lo-Dash function helpers, one advantage of using wrap() is that the caller of the generated function can supply more data via arguments, as demonstrated in the following code:

```javascript
function strong(value) {
    return '<strong>' + value + '</strong>;
}

function regex(exp, val) {
    exp = _.isRegExp(exp) ?
        exp : new RegExp(exp);
    return _.isUndefined(val) ?
        exp : exp.exec(val);
}

var boldName = _.wrap('Marianne', strong),
    getNumber = _.wrap('(\d+)', regex);

boldName();
// → "<strong>Marianne</strong>"

getNumber('abc123')[1];
// → "123"
```

The first function, strong(), wraps the value in <strong/> tags. The second function, regex(), is a little more involved. It wraps a value in a RegExp instance. But it's smart enough to do this only if the value is a string—if it's already a regular expression, there's no need to create another. Also, if a value is supplied to the second argument, it'll execute the regular expression against it, returning the result.
Working with Functions

The result of calling `boldName()` is self-explanatory. The value 'Marianne' is wrapped with the `strong()` function. The `getNumber()` function is a result of wrapping a regular expression string that looks for numbers. However, the call to `getNumber()` supplies an additional argument, that is, the call provides a string to execute the regular expression against it. This is why we access the result using a numerical index following the call.

Let's turn our attention to decorating existing functions with new functionality using `wrap()`:

```javascript
var user = _.sample(['Scott', 'Breanne']);

var allowed = ['Scott', 'Estelle'];

function permission(func) {
    if (_.contains(allowed, user)) {
        return func.apply(null, _.slice(arguments, 1));
    }
    throw new Error('DENIED');
}

function echo(value) {
    return value;
}

var welcome = _.wrap(echo, permission);

welcome('Yo there!');
```

The basic idea here is to decorate the `echo()` function with permission checking ability. The `permission()` function will call the function that is passed to it only if the `user` variable exists in the `allowed` array. An exception is raised if this is not the case. Repeatedly running this code will randomly generate denied errors. It all depends on whether 'Breanne', who isn't in the `allowed` array, is sampled as the current user or not.
Function constraints

Similar to decorating functions with new behavior are the constraints imposed on functions. This impacts when and how often the function can be called. Function constraints also control how values returned by calling a function are cached. Lo-Dash has functions that deal with each of these scenarios.

Limiting call counts

There are two Lo-Dash functions that deal with counting the number of times a given function is called. The `after()` function will execute a callback after the composed function is called a given number of times. The `once()` function constrains the given function to being called only once. Let's look at `after()` and see how it works:

```javascript
function work(value) {
  progress();
}

function reportProgress() {
  console.log(++complete + '%');
  progress = complete < 100 ?
    _.after(0.01 * collection.length, reportProgress) :
    _.noop;
}

var complete = 0,
  collection = _.range(9999999),
  progress = _.noop;

reportProgress();

_.forEach(collection, work);
// →
// 1%
// 2%
// 3%
// ...
```
Working with Functions

The `work()` function is a contrived function that actually does nothing other than calling `progress()`, which notifies the world that progress has been made. A real function that actually did work would call `progress()` after having done the work.

Next, we have a `reportProgress()` function. It is responsible for logging the progress. It also creates the `progress()` function using `after()`. Until the `complete` variable has reached 100 percent, it'll call `reportProgress()` again, which redefines the `progress()` function. The `after()` function will call the callback function supplied to it after the `progress()` function has been called `x` number of times. In this case, `x` is 1 percent of the collection length.

To sum up, `reportProgress()` defines the `progress()` function. This function is called by worker functions that need to notify the world about their progress. After `progress()` has been called so many times, `reportProgress()` is called. This is where the progress is logged and `progress()` is redefined.

All this is put into action by creating a rather large collection and iterating over it, calling `work()` along the way. But before the iterating starts, we kick off the progress tracker by calling `reportProgress()`. One nice aspect of this code is that there is a separation of concerns between tracking progress and performing work. The worker function only needs to worry about calling `progress()`. The `reportProgress()` is only concerned about periodically logging the progress and doesn't care about the actual work being done.

Asynchronous operations can make use of `after()` as well. The previous example explicitly called the function that was created by `after()`. However, what if we want to synchronize what happens after several asynchronous callback functions have fired? Let's find out using the following code:

```javascript
function process(coll, callback) {
    var sync = _.after(coll.length, callback);
    _.forEach(coll, function() {
        setTimeout(sync, _.random(2000));
    });
    console.log('timeouts all set');
}

process(_.range(5), function() {
    console.log('callbacks completed');
});
// →
// timeouts all set
// callbacks completed
```
First, we have a `process()` function meant to symbolize a long-running asynchronous process—something that runs in the background, in other words. This function takes two arguments: a collection and a callback. The `callback` is a function that is called after each asynchronous operation on the collection has completed. We do this by creating a new `sync()` function using `after()`. The collection length is passed to `after()`. This means that after `sync()` has been called five times, which is the length of our collection, the callback is called.

Next, we randomly choose a timeout and call `sync()`—this is the asynchronous part. After all the timeouts have been set, we then log that the calls to `sync()` have been scheduled. The callback that executes when these are done logs a basic message.

Sometimes, it's useful to call a function just once. Beyond that, it is just useless repetition—harmless, but unnecessary. Therefore, a useful constraint for a function might be only allowing it to be called once. But how would we enforce such a thing? This can be done using the following code:

```javascript
function getLeader(coll) {
  return _.first(_.sortBy(coll, 'score').reverse());
}

var collection = [
  { name: 'Dana', score: 84.4 },
  { name: 'Elsa', score: 44.3 },
  { name: 'Terrance', score: 55.9 },
  { name: 'Derrick', score: 86.1 }
];

var leader = _.once(getLeader);

leader(collection);
// → { name: "Derrick", score: 86.1 }
```

The `getLeader()` function in this code takes a collection and returns the name of the leader, according to the `score` property. We use this function to construct the `leader()` function. Using `once()`, we tell the `leader()` function to only call `getLeader()` once, and only once. You can't prevent the caller from making 50 calls to these functions. The job of the `once()` function is to encapsulate the function passed to it, storing the return value of the first invocation. If this value is set, it's cached for subsequent calls. So the preceding code assumes that the collection is unchanging and the leader will always be the same.
Caching values
The preceding example gave the first glimpse into caching values with Lo-Dash. If the function is constrained to be called only once, it might as well store the value of that first invocation. This is almost caching as a side effect—there's a more explicit approach that uses the `memoize()` function. Explicit caching is especially useful for mathematical functions, where given the same input, the same output is always produced. This is also referred to as referential transparency. An example for this is as follows:

```javascript
function toCelsius(degrees) {
    return (degrees - 32) * 5 / 9;
}

function toFahrenheit(degrees) {
    return degrees * 9 / 5 + 32;
}

var celsius = _.memoize(toCelsius),
    fahrenheit = _.memoize(toFahrenheit);

toCelsius(89).toFixed(2) + ' C';
// → "31.67 C"

celsius(89).toFixed(2) + ' C';
// → "31.67 C"

toFahrenheit(23).toFixed(2) + ' F';
// → "73.40 F"

fahrenheit(23).toFixed(2) + ' F';
// → "73.40 F"
```

Here, we have two simple mathematical functions and they are good candidates for memoization. The `toCelsius()` function takes the given degrees in Fahrenheit and returns the Celsius equivalent. The `toFahrenheit()` function is the inverse—it takes a Celsius argument and returns a Fahrenheit value. We then take these two functions and wrap them with `memoize()`, yielding two new functions, `celsius()` and `fahrenheit()`.
After that, we make two calls each to the same function successively. The first call computes the value and stores it. The second call returns the cached result and computes nothing, but how does this cache lookup work? How does the memoized function know to use a value from the cache and not to compute a value? Let's find this out using the following code:

```javascript
function toCelsius(degrees) {
  return (degrees - 32) * 5 / 9;
}

function toFahrenheit(degrees) {
  return degrees * 9 / 5 + 32;
}

function convertTemp(degrees, system) {
  return system.toUpperCase() === 'C' ?
    toFahrenheit(degrees).toFixed(2) + ' F' :
    toCelsius(degrees).toFixed(2) + ' C';
}

var convert = _.memoize(convertTemp, function(degrees, system) {
  return degrees + system;
});

convert(89, 'F');
convert(89, 'F');
convert(23, 'C');
convert(23, 'C');
```

By default, the resulting function generated by `memoize()` will use the first supplied argument as the cache key. The cache is a simple object and values are looked up by the property key. In the previous example, the memoized functions accepted only one argument. This is fine, but in more complex functions that accept more than one argument, you need a means to resolve the lookup key, as is illustrated in the preceding example.

This is basically a rewrite of the previous example, as it generates the same result. We still have the `toCelsius()` and `toFahrenheit()` functions, but we've introduced a new `convertTemp()` function. This function accepts two arguments: the degrees and the temperature system these degrees represent. Based on these argument values, we can make the appropriate call to either `toCelsius()` or `toFahrenheit()`.
We then construct the `convert()` function, a memoized version of `convertTemp()`. You'll notice the second function passed to `memoize()` here builds and returns a cache key. Without it, cache keys would still be consulted based only on the first argument value, which would return incorrect data. Be careful.

You may have noticed that we could have continued using the previously cached functions, `celsius()` and `fahrenheit()`. That would mean a multilayered cache, which sounds kind of cool actually. Resist the temptation to do stuff like this. If you're performing badly enough to require a multilayer cache, it's time to reconsider the design at a higher level.

**Timed execution**

By nature, JavaScript code executes synchronously, that is, you don't have multiple threads of control, each running a piece of your code and competing for the CPU's attention. There are web workers in modern browsers, but these are far from commonplace yet and don't share much similarity with a threading API you'd find in another language. The upside to all of this is that you, as the programmer, don't need to concern yourself with synchronization primitives and all the other nasty details associated with multithreaded programming.

Instead, you face a different kind of difficulty in that you have to deal with events, the DOM, and other forms of callbacks; so much for synchronous code. Sometimes, this is actually desired. For example, you need to wait for a predetermined amount of time before something can happen. Or, perhaps you want to update the DOM and then pick up where you left off. Lo-Dash has tools that help you figure out the tricky details when it comes to timing function calls and coping with the side effects.

**Delaying function calls**

The `delay()` function is used to execute a given callback function after the given number of milliseconds has elapsed. This actually works the same way as the built-in `setTimeout()` function does. This is shown in the following code:

```javascript
function poll() {
    if (++cnt < max) {
        console.log('polling round ' + (cnt + 1));
        timer = _.delay(poll, interval);
    } else {
```
var cnt = -1,
    max = 5,
    interval = 3000,
    timer;

poll();
// →
// polling round 1
// polling round 2
// polling round 3
// polling round 4
// polling round 5

This code defines a `poll()` function that is used to periodically log which round of polling we're on. Polling is a common pattern used in frontends to synchronize data from the API, with what the user is looking at. We've set the `max` variable, which controls the number of polling iterations, to 5. The `interval` variable is set to 3000 milliseconds. It controls the polling call frequency. You can see that the `poll()` function will first check whether we've already reached the maximum number of iterations or not. If not, the `timer` variable gets a timeout value—just an integer—by calling `delay()`. The `delay()` callback is `poll()`. If we've already reached our threshold, the timeout is cleared and there's no further poll scheduling.

If you look closely, you'll notice that there's no difference between using `delay()` and the built-in `setTimeout()` function. Both accept a callback function and duration as arguments, and both return a timeout number that can be cleared using `clearTimeout()`. What's interesting about `delay()` compared to `setTimeout()` is how they deal with arguments. Let's see how arguments are handled:

```javascript
function sayHi(name, delay) {
    function sayHiImp(name) {
        console.log('Hi, ' + name);
    }
    if (_.isUndefined(delay)) {
        _.delay(sayHiImp, 1, name);
    } else {
        _.delay(sayHiImp, delay, name);
    }
}
sayHi('Jan');
```
Working with Functions

```javascript
sayHi('Jim', 3000);
// →
// Hi, Jan
// Hi, Jim
```

Here we've created a `sayHi()` function. This has a nested function within the called `sayHiImp()` function, which is the actual implementation. The `sayHi()` function is just a wrapper for `sayHiImp()`. It logs the given `name` parameter and checks whether the `delay` parameter was supplied or not; if not, it supplies a default `delay` value. It's important that our function either always runs synchronously or asynchronously, but never both. However, if there's a `delay` value, we use it with the `delay()` function to postpone the call to `sayHiImp()`. Notice that we pass the `name` parameter to the `delay()` call as well. Rather than having to construct our own partial function, we let `delay()` make one for us.

### Deferring function calls

Whenever JavaScript code is run in the browser, it kicks off what is known as a call stack. Most programming languages share the notion of a call stack. It can be thought of as a traceable graph of function calls, starting with the root call. What's interesting is that the JavaScript call stack and the DOM are two completely separate entities that share the same thread of control. The implication is that the DOM doesn't run while there's an active JavaScript call stack. This is why long-running JavaScript code locks up UI interactivity.

Using the `defer()` function is a workaround for scenarios where you have a function that could take a while (a while being a relative term here—2 seconds is a while). You can push the call to that function till after the call stack has cleared, as shown in the following code:

```javascript
function expensive() {
    _.forEach(_.range(Math.pow(2, 25)), _.noop);
    console.log('done');
}

_.defer(expensive);
console.log('computing...');
// →
// computing...
// done
```
The `expensive()` function does nothing but hog the CPU for a bit, preventing the `console.log()` call from running. So we use `defer()` to call `expensive()`, which waits till the current call stack has finished. The 'computing...' string is logged as the last statement in the call stack. Shortly thereafter, the 'done' string appears in the console log. The trick is that we're giving the DOM a chance to update before the expensive code runs.

An alternative approach to calling `defer()` every time you want to invoke something after the call stack has cleared is to create a wrapper function. You then call this wrapper as you would call any other function and it'll take care of deferring it for you. This is done using the following code:

```javascript
function deferred(func) {
    return _.defer.apply(_, ([ func ]).concat(_.slice(arguments, 1)));
}

function setTitle(title) {
    console.log('Title: "' + title + ""');
}

function setState(app) {
    console.log('State: "' + app.state + '"');
}

var title = _.wrap(setTitle, deferred),
    state = _.wrap(setState, deferred),
    app = { state: 'stopped' };

title('Home');
setState(app);
app.state = 'started';
// →
// Title: "Home"
// State: "started"
```

There are two functions here, `setTitle()` and `setState()`, both of which we'd like to be made deferrable. The first function takes a `title` argument and logs it. The second function takes an `app` object and logs the `state` property of that object. The `deferred()` function is a wrapper. We'll use it along with `wrap()` to make any function deferrable. All `deferred()` does is apply `defer()` to the function that was passed along with some arguments.
Next, you can see that the `title()` function is the deferred version of `setTitle()` while the `state()` function is the deferred version of `setState()`. We also have an `app` object with an initial state of 'stopped'. Calling `title()` and `state()` will always be deferred to after the call stack clears. This point is further illustrated in the preceding code by setting the state to `started`, after the call to `state()`. You can guess which string is logged.

**Throttling function calls**

Often, events in the DOM can trigger much more frequently than you're equipped to handle them. The simple act of moving the mouse pointer around has the potential to generate hundreds of events per second. If each of these events has a handler and that handler does anything meaningful, the UI will lag. There's simply no way to keep up, no matter how fast the processor is. The only way to keep up is to ignore the majority of these events and only responds at a certain frequency. The idea is illustrated in the following code:

```javascript
var el = document.querySelector('#container'),
onMouseMove = _.throttle(function(e) {
    console.log('X: ' + e.clientX + ' Y: ' + e.clientY);
}, 750);

el.addEventListener('mousemove', onMouseMove);
window.addEventListener('hashchange', function cleanup() {
    el.removeEventListener('mousemove', onMouseMove);
    window.removeEventListener('mousemove', cleanup);
});
```

The `el` variable is a DOM element that we want to listen to for `mousemove` events. The `onMouseMove` function is created by passing a function to `throttle()`. This callback simply logs the mouse coordinates. We also pass 750 to `throttle()` as the maximum frequency with this callback is allowed to run. Next, we bind the event and set up the cleanup actions to remove the listener when we're done with it. Had we not throttled `onMouseMove()`, you would see a noticeable difference in the `console.log()` verbosity.
Debouncing function calls

Debouncing functions is similar to throttling them. The difference is in what happens when the wait duration has elapsed. With `throttle()`, the function is invariably called. For example, if the `wait` value was set to 10 milliseconds on a throttled function, and the function was called during those 10 milliseconds, it'll get called before the next wait. With `debounce()`, during the 10-millisecond wait, if the function was called, it'll wait an additional 10 milliseconds. Let's look at some debouncing code:

```javascript
function log(msg, item) {
    console.log(msg + ' ' + item);
}

var debounced = _.debounce(_.partial(log, 'debounced'), 1),
    throttled = _.throttle(_.partial(log, 'throttled'), 1),
    size = 1500;

_.forEach(_.range(size), debounced);
_.forEach(_.range(size), throttled);
```

We have a simple `log()` function that logs a message and an item number. We then proceed to build a `debounced()` and a `throttled()` version of the function. Then we run both through the same-sized loop. What's the difference? The output looks something like this:

```
throttled 0
throttled 1
throttled 744
debounced 1499
throttled 1499
```

What happened here? We set the `wait` time to 1 millisecond for both `debounced()` and `throttled()`. In the time it took to process 1500 items, the wait period elapsed twice for the `throttled()` function. As soon as that happened, the `log()` function was called, hence the output. Notice that the `debounce()` output happened only after the processing was done. That's because `debounce()` was called many times during the 1-millisecond wait, and again during the next wait.
The `throttle()` function actually uses `debounce()` under the hood. All of the complexity is in `debounce()` and it accepts several configuration options. Among these are the leading and trailing edges of execution. What does this mean? You'll notice in the preceding output that the `throttled()` function is called after `debounce()`. That's the trailing edge of the wait period. The leading edge of the wait period is before the wait period starts. Both of these edges default to `true` for `throttle()`. This means that you're in an intense loop where your throttled function is being hammered, the function is called immediately, before waiting for the next call. Then, if the loop ends abruptly, the function is called again when the wait period ends.

### Composing and currying functions

The last section of this chapter is about assembling functions that realize larger behavior out of smaller functions. There are two ways to assemble such functions. The first is to use the appropriately named `compose()` function, which performs a nested invocation of the provided functions, or where order is important, we can use the `flow()` function to return values together. Currying lets you adapt your function to be called successively in different contexts. Each of these Lo-Dash tools lets you take the existing functionality in your application and build on it in interesting ways.

#### Composing functions

The `compose()` function builds a new function out of the provided functions. When we call this new function, a nested invocation of the supplied function starts, that is, the last supplied function is called with any additional arguments. The returned value is then fed to the next function and so on, ultimately producing a value for the caller. This is better explained in the following example:

```javascript
function dough(pizza) {
  if (_.isUndefined(pizza)) {
    pizza = {};
  }
  return _.extend({
    dough: true
  }, pizza);
}

function sauce(pizza) {
  if (!pizza.dough) {
```
throw new Error('Dough not ready');
}
return _.extend({
  sauce: true
}, pizza);
}

function cheese(pizza) {
  if (!pizza.sauce) {
    throw new Error('Sauce not ready');
  }
  return _.extend({
    cheese: true
  }, pizza);
}

var pizza = _.compose(cheese, sauce, dough);
pizza();
// → { cheese: true, sauce: true, dough: true }

There are three functions responsible for assembling pizza—dough(), sauce(), and cheese(). The job of each one of these functions is to set their corresponding attribute to true on the supplied pizza object. The pizza() function is composed using these functions which in turn use the compose() function. So calling pizza() will call cheese(sauce(dough())). Note some of the checking that happens in these functions. For example, dough() will accept an object or construct a new one. However, the sauce() function won't work if there's no dough attribute. Likewise, cheese() complains if there's no sauce.

While being able to compose functions is handy, it's a good idea to have precondition checking. Then they fail fast, so other developers attempting to compose something out of your functions have an obvious indication if something isn't possible.

If the reverse order of the function invocation is confusing, don't worry. We can reverse the order using the flow() function. Using the same pizza functions, we could make a slight modification to the pizza() composition function:

var pizza = _.flow(dough, sauce, cheese);

return pizza();
The compose() function is actually an alias for the flowRight() function. The flow() and flowRight() functions are newer. In previous versions of Lo-Dash, the compose() function was standalone.

Currying functions

Have you ever found yourself having to create a bunch of variables that do nothing aside from eventually getting passed to a function? Instead of variable creation, the currying technique lets you partially apply the function. That is, you call the function, supplying only the data you have at that moment. Curried functions will keep returning the function until it has all the arguments necessary. This technique is explained using the following example:

```javascript
function makePizza(dough, sauce, cheese) {
    return {
        dough: dough,
        sauce: sauce,
        cheese: cheese
    };
}

function dough(pizza) {
    return pizza(true);
}

function sauceAndCheese(pizza) {
    return pizza(true, true);
}

var pizza = _.curry(makePizza);

sauceAndCheese(dough(pizza));
// → { cheese: true, sauce: true, dough: true }
```

The makePizza() function has any arity of three—the number of arguments expected by the function. This means that the pizza() function created by calling curry() on makePizza() will keep returning the function until it's invoked with three arguments. We have the flexibility to pass these arguments however we want. This could be all three at once or it could be one at a time. This means that different contexts could pass data into the function, without the need to store them elsewhere.
Summary

Hopefully after reading this chapter, your appreciation for functions in JavaScript went up a little. Lo-Dash just makes functional programming in the frontend that much better. Functions in JavaScript are flexible by default, changing the execution context for example. This chapter showed you how some Lo-Dash functions make working with function contexts much easier by removing much of the boiler-plate code that would otherwise be needed. Partials are fundamental to functional programming, but it's one of those tasks that's anything but easy in JavaScript. Lo-Dash makes it easy to create partials and to decorate functions by wrapping them with additional logic.

We looked at functions that help constrain when a function should run. For example, should a function be allowed to run only once? Should the return values be cached? Timing the execution of functions is a complex topic, especially when you consider the DOM and how it integrates with the JavaScript call stack. Lo-Dash has a number of functions that deal with managing the timed execution of functions. We looked at these in detail.

The chapter wrapped up with a look at how to compose larger pieces of functionality out of smaller functions. Currying functions let you define functions flexible enough to be invoked in a number of contexts, reducing the need to temporarily store arguments before they're passed. And on that note, we covered the Lo-Dash fundamentals. The concepts you've learned so far about collections, objects, and functions are applicable throughout the remainder of the book. We're now ready to move on to mapping and reducing values, a powerful technique that you'll utilize over and over again when programming with Lo-Dash.
Where to buy this book

You can buy Lo-Dash Essentials from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.