CHAPTER 9

Programming Without Class

When people are first exposed to JavaScript and its minimal set of tools (functions, objects, prototypes, and arrays), many are underwhelmed. Therefore, in order to "modify" JavaScript to conform to their idea of what it takes to model software solutions, they very often seek out or re-create class-based systems using the primordial ooze. This desire is completely understandable given that in general people will often seek the familiar. However, since you've come this far in exploring functional programming in JavaScript, it's worth tying all of the threads from the previous chapters into a coherent exploration of how to reify functional and object-oriented thinking.

This chapter starts by walking the path of data and function thinking that I've talked about throughout the book. However, while thinking in functions and simple data is important, there will come a time when you may need to build custom abstractions. Therefore, I will cover a way to "mix" discrete behaviors together to compose more complex behaviors. I will also discuss how a functional API can be used to hide such customizations.

Data Orientation

Throughout the course of this book, I've intentionally defined my data modeling needs in terms of JavaScript primitives, arrays, and objects. That is, I've avoided creating a hierarchy of types in favor of composing simple data together to form higher-level concepts like tables (Chapter 8) and commands (Chapter 4). Adhering to a focus on functions over methods allowed me to provide APIs that do not rely on the presence of object thinking and methodologies. Instead, by adhering to the functional interfaces, the *actual* concrete types implementing the data abstractions mattered less. This provided flexibility to change the implementation details of the data while maintaining a consistent functional interface.

Figure 9-1 illustrates that when using a functional API, you don't really need to worry about what types are flowing between the nodes in a call chain.

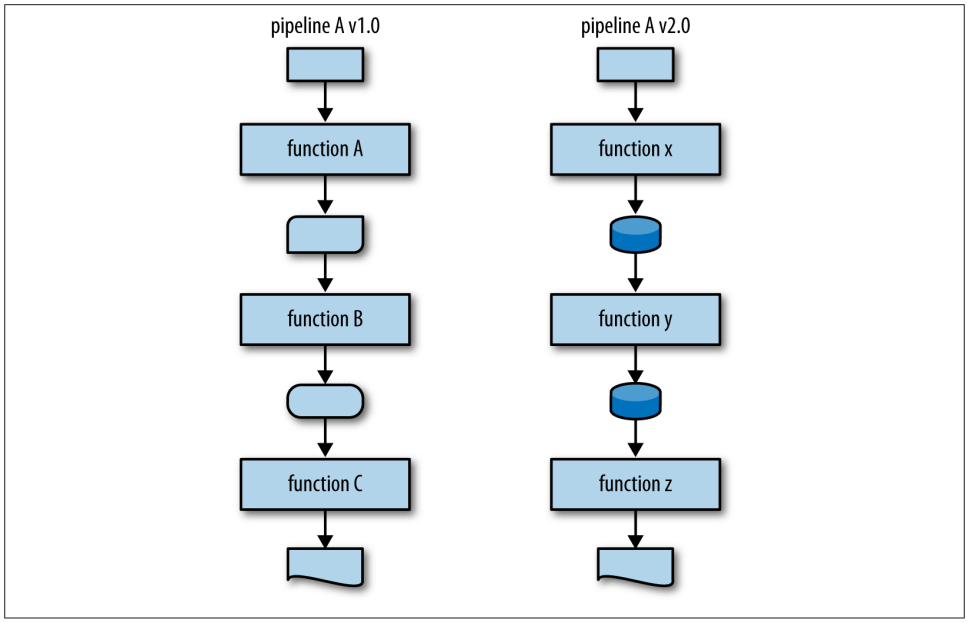

Figure 9-1. When adhering to a functional interface, the type of intermediate data matters little and can evolve (or devolve) as needed especially if you're concerned primarily with the beginning and end of a computation

Of course, the functions themselves should be able to handle the types flowing between, but well-designed APIs are meant to compose and should abstract the details of intermediate types. However, there are times when object-centric thinking is crucial. For example, the `LazyChain` implementation from Chapter 8 specifically deals with the lazy execution of methods on a target object. Clearly, the very nature of the problem leads to a solution where methods are called on some object. However, the implementation requires that the user of `LazyChain` deal directly with the creation of instances of that type. Thanks to JavaScript's extreme flexibility, there is no need to create a specialized `LazyChain` type. Instead, a lazy chain is whatever is returned from a function `lazyChain` responding to `.invoke` and `.force`.

```
function lazyChain(obj) {
  var calls = [];

  return {
    invoke: function(methodName /* args */) {
      var args = _.rest(arguments);
```

```
      calls.push(function(target) {
        var meth = target[methodName];

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

      return this;
    },
    force:  function() {
      return _.reduce(calls, function(ret, thunk) {
        return thunk(ret);
      }, obj);
    }
  };
}
```

This is almost the exact code as in the implementation of `LazyChain` except for the following:

- The lazy chain is initiated via a function call.
- The call chain (in `calls`) is private data.[1]
- There is no explicit `LazyChain` type.

The implementation of `lazyChain` is shown here:

```
var lazyOp = lazyChain([2,1,3])
  .invoke('concat', [7,7,8,9,0])
  .invoke('sort');

lazyOp.force();
//=> [0, 1, 2, 3, 7, 7, 8, 9]
```

There are certainly times to create explicit data types, as I'll show in the next section, but it's good to defer their definition until absolutely necessary. Instead, a premium is placed on programming to abstractions. The idea of how to interact with a lazy chain is more important than a specific `LazyChain` type.

JavaScript provides numerous and powerful ways to defer or eliminate the need to create named types and type hierarchies, including:

- Usable primitive data types
- Usable aggregate data types (i.e., arrays and objects)
- Functions working on built-in data types

1. That the `chain` array is private slightly complicates the ability to chain lazy chains with other lazy chains. However, to handle this condition requires a change to `force` to identify and feed the result of one lazy chain to the next.

- Anonymous objects containing methods
- Typed objects
- Classes

Graphically, the points above can be used as a checklist for implementing JavaScript APIs, as shown in Figure 9-2.

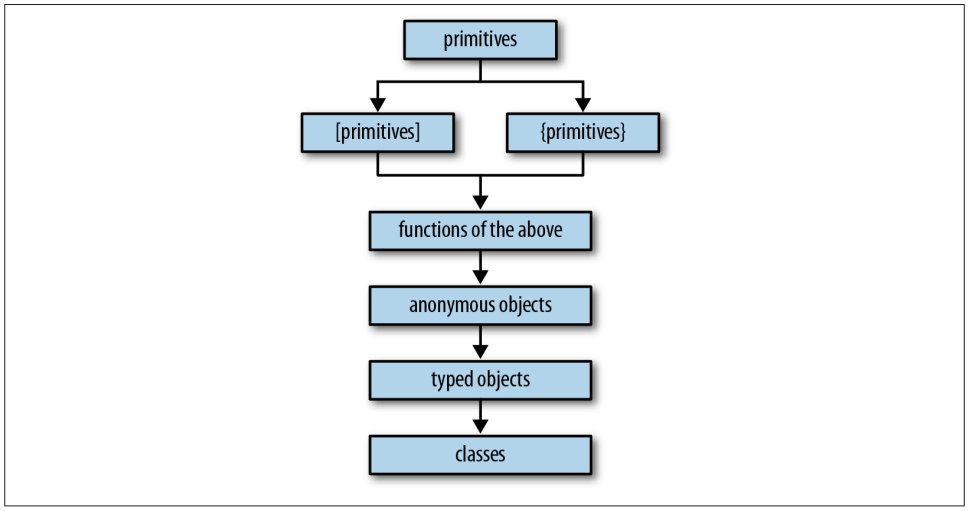

Figure 9-2. A "hierarchy" of data thinking

Very often, JavaScript developers will invert the hierarchy shown in Figure 9-2 and start immediately with constructing classes, thus blowing their abstraction budget from the start. If you instead choose to start with built-in types coupled with a fluent, functional API, then you allow yourself a lot of flexibility for expansion.

Building Toward Functions

For most programming tasks, the activities happening in the middle of some computation are of primary importance (Elliott 2010). Take, for example, the idea of reading in a form value, validating it, performing operations on the new type, and finally sending the new value somewhere else as a string. The acts of getting to and from a string are small compared to the validation and processing steps.

At the moment, the tools that I've created to fulfill this kind of task are a mix of functional and object-based thinking. However, if I factor toward functions only, then a fluent solution can evolve.

First of all, the lazy chains are clearly object-centric and in fact require the stringing of methods to operate. However, lazy chaining can be deconstructed into three stages:

1. Acquire some object.
2. Define a chain in relation to the object.
3. Execute the chain.

The act of acquiring an object is trivial; it simply occurs as part of running JavaScript code. Defining a chain, however, is where it gets interesting. Whereas a lazy chain is defined in terms of the actions to perform on a specific instance, by lifting its creation into a function, I can make lazy operations generic across types of objects instead:

```
function deferredSort(ary) {
  return lazyChain(ary).invoke('sort');
}
```

This allows me to create lazy sorts on any array via a regular function call:

```
var deferredSorts = _.map([[2,1,3], [7,7,1], [0,9,5]], deferredSort);

//=> [<thunk>, <thunk>, <thunk>]
```

Naturally, I'd like to execute each thunk, but since I'm factoring to functions, I'd prefer to encapsulate the method call:

```
function force(thunk) {
  return thunk.force();
}
```

Now I can execute arbitrary lazy chains:

```
_.map(deferredSorts, force);

//=> [[1,2,3], [1, 7, 7], [0, 5, 9]]
```

And now that I've "lifted" the method calls into the realm of functional application, I can define discrete chunks of functionality corresponding to the atoms of data processing:

```
var validateTriples  = validator(
  "Each array should have three elements",
  function (arrays) {
    return _.every(arrays, function(a) {
      return a.length === 3;
    });
  });

var validateTripleStore = partial1(condition1(validateTriples), _.identity);
```

Aggregating the validation into its own function (or many functions, perhaps) allows me to change validation independent of any of the other steps in the activity. Likewise, it allows me to reuse validations later for similar activities.

Double checking that the validation works as expected:

```
validateTripleStore([[2,1,3], [7,7,1], [0,9,5]]);
//=> [[2,1,3], [7,7,1], [0,9,5]])

validateTripleStore([[2,1,3], [7,7,1], [0,9,5,7,7,7,7,7,7]]);
// Error: Each array should have three elements
```

Now I can also define other processing steps that are (not necessarily) lazy:

```
function postProcess(arrays) {
  return _.map(arrays, second);
}
```

Now I can define a higher-level activity that aggregates the pieces into a domain-specific activity:

```
function processTriples(data) {
  return pipeline(data
            , JSON.parse
            , validateTripleStore
            , deferredSort
            , force
            , postProcess
            , invoker('sort', Array.prototype.sort)
            , str);
}
```

The use of `processTriples` is as follows:

```
processTriples("[[2,1,3], [7,7,1], [0,9,5]]");

//=> "1,7,9"
```

The nice part about adding validations to your pipelines is that they will terminate early when given bad data:

```
processTriples("[[2,1,3], [7,7,1], [0,9,5,7,7,7,7,7,7]]");

// Error: Each array should have three elements
```

This allows me to now use this function anywhere that such a pipeline of transformations might be appropriate:

```
$.get("http://djhkjhkdj.com", function(data) {
  $('#result').text(processTriples(data));
});
```

You could make this process more generic by abstracting out the reporting logic:

```
var reportDataPackets = _.compose(
  function(s) { $('#result').text(s) },
  processTriples);
```

Exploring the use of `reportDataPackets` is as follows:

```
reportDataPackets("[[2,1,3], [7,7,1], [0,9,5]]");
// a page element changes
```

Now you can attach this discrete behavior to your application to achieve a desired effect:

```
$.get("http://djhkjhkdj.com", reportDataPackets);
```

Creating functions in general allows you to think about problems as the gradual transformation of data from one end of a pipeline to another. As you'll recall from Figure 9-1, each transformation pipeline can itself be viewed as a discrete activity, processing known data types in expected ways. As shown in Figure 9-3, compatible pipelines can be strung end to end in a feed-forward manner, while incompatible pipelines can be linked via adapters.

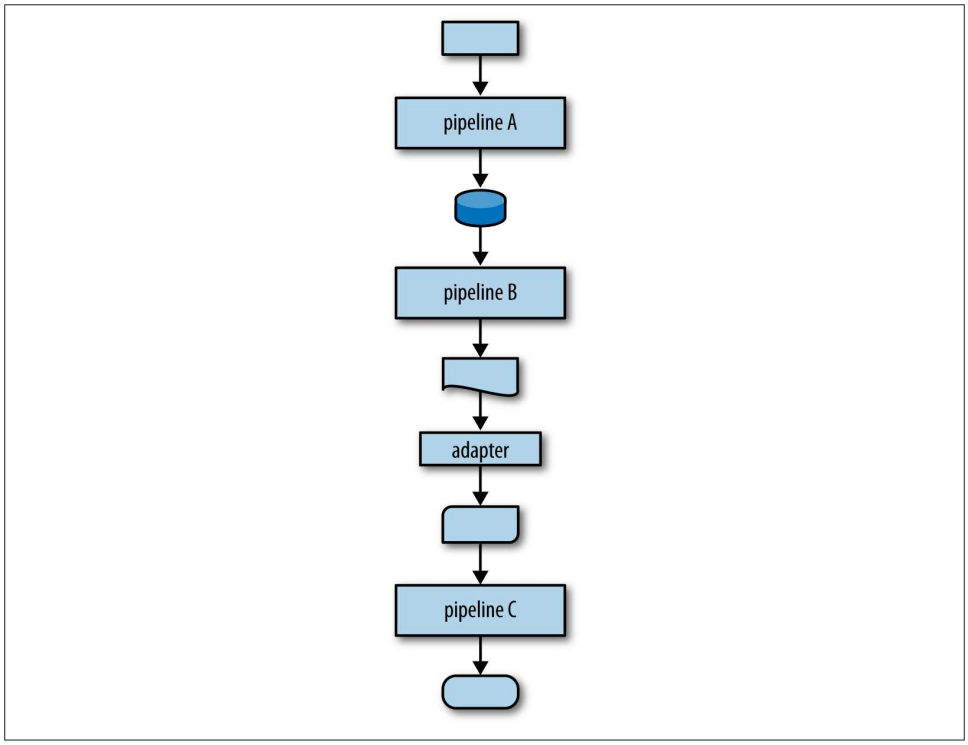

Figure 9-3. Linking pipelines directly or via adapters

From a program-wide perspective, pipelines with adapters can be attached to input and output sources. This type of thinking allows you to compose a system from smaller, known parts, while allowing the flexibility to interchange pieces and intermediate data representations as needed in the future. The idea of data flowing through transformers is a scalable notion, from the level of a single function up to the level of whole systems.

However, there are times when object-level thinking is appropriate, especially when concrete types adhering to generic mixins are the right abstraction. In the next section, I'll talk about the idea of a mixin and how it can be used to build up toward functional abstractions.

Mixins

While I've spent a lot of time and pages outlining a functional style of programming, there are times when objects and methods are just the right solution. In this section, I will outline an approach called mixin-based extension, which is similar to the way that class-based systems are built but intentionally constrained. Before diving into mixins directly, let me take a moment to motivate the need for object-thinking. Imagine a function `polyToString` that takes an object and returns a string representation of it. A naive implementation of `polyToString` could be written as follows:

```
function polyToString(obj) {
  if (obj instanceof String)
    return obj;
  else if (obj instanceof Array)
    return stringifyArray(obj);

  return obj.toString();
}

function stringifyArray(ary) {
  return ["[", _.map(ary, polyToString).join(","), "]"].join('');
}
```

As shown, the initial implementation of `polyToString` can be written as nested `if` statements where each branch checks the type. The addition of `stringifyArray` is added to create nicer looking string representations for strings. Running through a few tests shows `polyToString` in action:

```
polyToString([1,2,3]);
//=> "[1,2,3]"

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

Seems reasonable, no? However, attempting to create the representation requires that I add a new `if` branch into the body of `polyToString`, which is kind of silly. A better approach might be to use something like `dispatch` from Chapter 5, which takes some

number of functions and attempts to execute each, returning the first non-`undefined` value:

```
var polyToString = dispatch(
  function(s) { return _.isString(s) ? s : undefined },
  function(s) { return _.isArray(s) ? stringifyArray(s) : undefined },
  function(s) { return s.toString() });
```

Again, the types are checked, but by using `dispatch`, I've at least abstracted each check into a separate function and have opened the door to further composition for extension purposes. Of course, that the use of `dispatch` works as expected is a nice bonus also:

```
polyToString(42);
//=> "42"

polyToString([1,2,[3,4]]);
//=> "[1, 2, [3, 4]]"

polyToString('a');
//=> "a"
```

As you might imagine, new types still present a problem if they do not already have a nice `#toString` implementation:

```
polyToString({a: 1, b: 2});
//=> "[object Object]"
```

However, rather than causing the pain of needing to modify a nested `if`, `dispatch` allows me to simply compose another function:

```
var polyToString = dispatch(
  function(s) { return _.isString(s) ? s : undefined },
  function(s) { return _.isArray(s) ? stringifyArray(s) : undefined },
  function(s) { return _.isObject(s) ? JSON.stringify(s) : undefined },
  function(s) { return s.toString() });
```

And again, the new implementation of `polyToString` works as expected:

```
polyToString([1,2,{a: 42, b: [4,5,6]}, 77]);

//=> '[1,2,{"a":42,"b":[4,5,6]},77]'
```

The use of `dispatch` in this way is quite elegant,[2] but I can't help but feel a little weird about it. Adding support for yet another type, perhaps `Container` from Chapter 7 can illustrate my discomfort:

```
polyToString(new Container(_.range(5)));

//=> {"_value":[0,1,2,3,4]}"
```

2. For the sake of expediency, I've delegated out to `JSON.stringify` since this section is not about converting objects to strings; nor, for that matter, is it about stringifying in general.

Certainly I could make this more pleasing to the eye by adding yet another link in the chain of calls composing `dispatch`, consisting of something like the following:

```
...
 return ["@", polyToString(s._value)].join('');
...
```

But the problem is that `dispatch` works in a very straightforward way. That is, it starts from the first function and tries every one until one of them returns a value. Encoding type information beyond a single hierarchical level would eventually become more complicated than it needs to be. Instead, an example like customized `toString` operations is a good case for method methodologies. However, accomplishing this goal is typically done with JavaScript in ways that go against the policies that I outlined in the Preface:

- Core prototypes are modified.
- Class hierarchies are built.

I'll talk about both of these options before moving on to mixin-based extension.

Core Prototype Munging

Very often, when creating new types in JavaScript, you'll need specialized behaviors beyond those composed or extended. My `Container` type is a good example:

```
(new Container(42)).toString();
//=> "[object Object]"
```

This is unacceptable. The obvious choice is that I can attach a `Container`-specific `toString` method onto the `prototype`:

```
Container.prototype.toString = function() {
  return ["@<", polyToString(this._value), ">"].join('');
}
```

And now all instances of `Container` will have the same `toString` behavior:

```
(new Container(42)).toString();
//=> "@<42>"

(new Container({a: 42, b: [1,2,3]})).toString();
//=> "@<{"a":42,"b":[1,2,3]}>"
```

Of course, `Container` is a type that I control, so it's perfectly acceptable to modify its `prototype`—the burden falls on me to document the expected interfaces and use cases. However, what if I want to add the ability to some core object? The only choice is to step on the core prototype:

```
Array.prototype.toString = function() {
  return "DON'T DO THIS";
```

```
}

[1,2,3].toString();
//=> "DON'T DO THIS"
```

The problem is that if anyone ever uses your library, then any array that she creates is tainted by this new `Array#toString` method. Therefore, for core types like `Array` and `Object`, it's much better to keep custom behaviors isolated to functions that are delegated to custom types. I did this very thing in `Container#toString` by delegating down to `polyToString`. I'll take this approach later when I discuss mixins.

Class Hierarchies

> In Smalltalk, everything happens somewhere else.
>
> —Adele Goldberg

When approaching the task of defining a system using a class-based object-oriented methodology, you typically attempt to enumerate the types of "things" that comprise your system and how they relate to one another. When viewing a problem through an object-oriented lens, more often than not the way that one class relates to another is in a hierarchical way. Say employees are kinds of people who happen to be either accountants, custodians, or CEOs. These relationships form an hierarchy of types used to describe the residents of any given system.

Imagine that I want to implement my `Container` type as a hierarchy of types (see Figure 9-4).

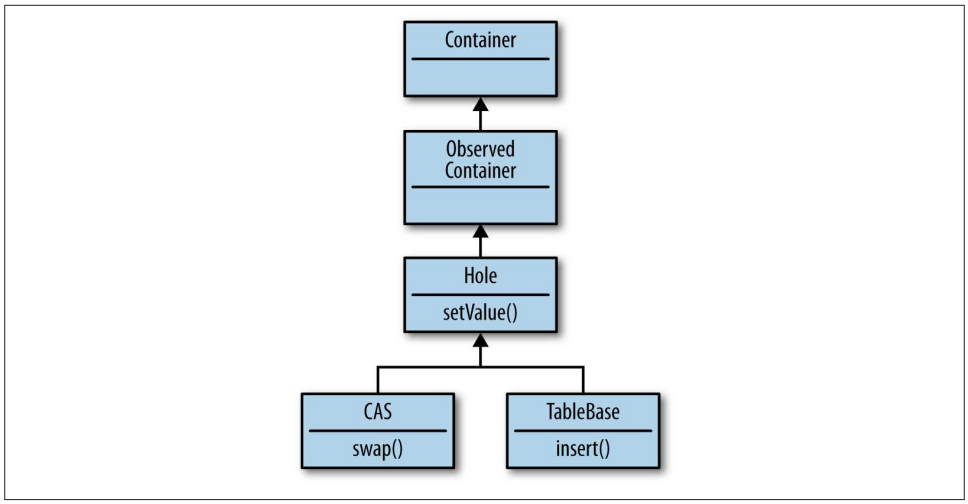

Figure 9-4. Representing Container types in a hierarchy

The diagram in Figure 9-4 states that at the root of the hierarchy is the `Container` type and from that derives `ObservableContainer`, which is used to attach functions that receive state change information. From `ObservableContainer`, I derive a `Hole` type that is "set-able." Finally, I define two different `Hole` types that have differing semantics for just how to assign values.

Using a stripped-down JavaScript class library based on a tiny library created by John Resig, I can sketch how this hierarchy might be constructed (Resig 2008):

```
function ContainerClass() {}
function ObservedContainerClass() {}
function HoleClass() {}
function CASClass() {}
function TableBaseClass() {}

ObservedContainerClass.prototype = new ContainerClass();
HoleClass.prototype = new ObservedContainerClass();
CASClass.prototype = new HoleClass();
TableBaseClass.prototype = new HoleClass();
```

Now that all of the hierarchical relationships are stitched together, I can test if they resolve as I expect:

```
(new CASClass()) instanceof HoleClass;
//=> true

(new TableBaseClass()) instanceof HoleClass;
//=> true

(new HoleClass()) instanceof CASClass;
//=> false
```

This is what I would expect—inheritance travels up the hierarchy, but not down. Now, putting some stubs in for implementation:

```
var ContainerClass = Class.extend({
  init: function(val) {
    this._value = val;
  },
});

var c = new ContainerClass(42);

c;
//=> {_value: 42 ...}

c instanceof Class;
//=> true
```

The `ContainerClass` just holds a value. However, the `ObservedContainerClass` provides some extra functionality:

```
var ObservedContainerClass = ContainerClass.extend({
  observe: function(f) { note("set observer") },
  notify: function() { note("notifying observers") }
});
```

Of course, the `ObservedContainerClass` doesn't do much on its own. Instead, I'll need a way to set a value and notify:

```
var HoleClass = ObservedContainerClass.extend({
  init: function(val) { this.setValue(val) },
  setValue: function(val) {
    this._value = val;
    this.notify();
    return val;
  }
});
```

As you might expect, the hierarchy is available to new `HoleClass` instances:

```
var h = new HoleClass(42);
// NOTE: notifying observers

h.observe(null);
// NOTE: set observer

h.setValue(108);
// NOTE: notifying observers
//=> 108
```

And now, at the bottom of the hierarchy, I start adding new behavior:

```
var CASClass = HoleClass.extend({
  swap: function(oldVal, newVal) {
    if (!_.isEqual(oldVal, this._value)) fail("No match");

    return this.setValue(newVal);
  }
});
```

A `CASClass` instance adds additional compare-and-swap semantics that say, "provide what you think is the old value and a new value, and I'll set the new value only if the expected old and actual old match." This change semantic is especially nice for asynchronous programming because it provides a way to check that the old value is what you expect, and did not change. Coupling compare-and-swap with JavaScript's run-to-completion guarantees is a powerful way to ensure coherence in asynchronous change.[3]

3. In a nutshell, run-to-completion refers to a property of JavaScript's event loop. That is, any call paths running during a particular "tick" of the event loop are guaranteed to complete before the next "tick." This book is not about the event-loop. I recommend David Flanagan's *JavaScript: The Definitive Guide*, 6th Edition, for a comprehensive dive into the JavaScript event system (and into JavaScript in general).

You can see it in action here:

```
var c = new CASClass(42);
// NOTE: notifying observers

c.swap(42, 43);
// NOTE: notifying observers
//=> 43

c.swap('not the value', 44);
// Error: No match
```

So with a class-based hierarchy, I can implement small bits of behavior and build up to larger abstractions via inheritance.

Changing Hierarchies

However, there is a potential problem. What if I want to add a new type in the middle of the hierarchy, called `ValidatedContainer`, that allows you to attach validation functions used to check that good values are used. Where does it go?

As shown in Figure 9-5, the logical place seems to be to put `ValidatedContainer` at the same level as `ObservedContainer`.

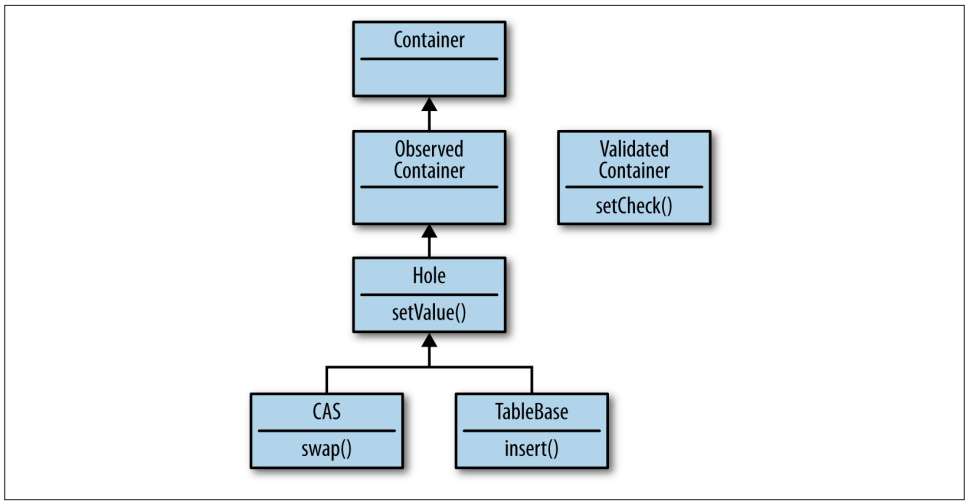

Figure 9-5. Extending the hierarchy

It's conceivable that I'd want all `Hole` instances to allow validation, but I don't really know that for certain (never mind the problem of multiple inheritance). I certainly do not want to assume that my users will want that behavior. What would be nice is if I could just extend it where needed. For example, if the `CAS` class needed validators, then

I could put `ValidatedContainer` above it in the hierarchy and just extend from it, as shown in Figure 9-6.

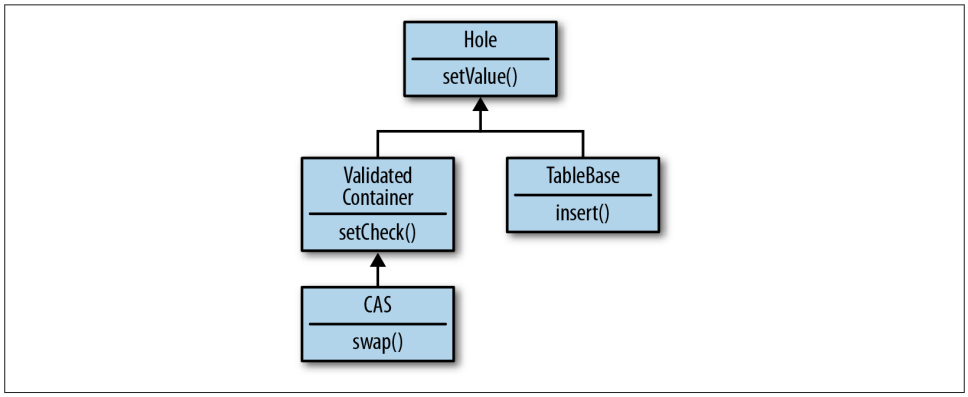

Figure 9-6. Moving a special-case class lower in the hierarchy is tricky

However, if a new type comes along that needs compare-and-swap semantics, but doesn't need validation, then the hierarchy in Figure 9-6 is problematic. I definitely shouldn't force that implementation to inherit from `CAS`.

The big problem with class hierarchies is that they are created under the assumption that we know the set of needed behaviors from the start. That is, object-oriented techniques prescribe that we start with a hierarchy of behaviors and fit our classes into that determination. However, as `ValidatedContainer` shows, some behaviors are difficult to classify ontologically. Sometimes behaviors are just behaviors.

Flattening the Hierarchy with Mixins

Let me try to simplify matters here. Imagine if I could take the base functionalities contained in `Container`, `ObservedContainer`, `ValidatedContainer`, and `Hole` and just put them all at the same level (see Figure 9-7).

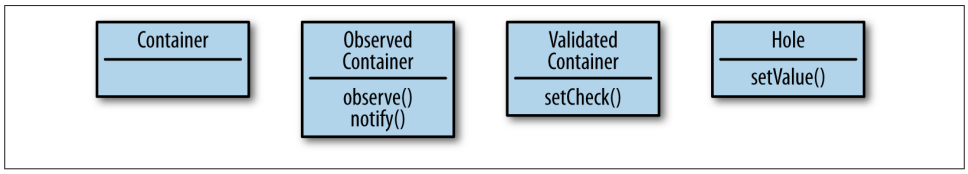

Figure 9-7. Flattening the hierarchy

If you blur your eyes a little, then Figure 9-7 shows that when flattening the hierarchy, what's left is not an implicit relationship between one type and another. In fact, the boxes

do not really define types at all. Instead, what they define are sets of discrete behaviors, or *mixins*. If all we have are behaviors, then the way to make new behaviors is to either define them anew or "mix" in existing behaviors.[4] This again hearkens back to the idea of composing existing functionality to create new functions.

So let me start anew with an implementation of `Container`:

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

Container.prototype.init = _.identity;
```

The implementation of this new `Container` constructor looks much like the implementation from Chapter 7, except this one has a call to an `init` method. The presence of the `init` call defines a mixin—or the means by which extension of the `Container` occurs in addition to the way that clients interact with it. Specifically, the mixin protocol for `Container` is as follows:

Extension protocol
: Must provide an `init` method

Interface protocol
: Constructor only

When designing APIs via mixin extension, you'll often need to delegate to unknown functions. This not only provides a standard for interacting with the types, but also allows extension points. In the case of `Container`, the `init` call delegates to Underscore's `_.identity`. Later on I will override `init`, but for now, see how `Container` is used:

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

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

So the new `Container` acts like the old. However, what I'd like to do is create a new type with similar, yet different behavior. The type that I have in mind, called a `Hole`, has the following semantics:

- Holds a value
- Delegates to a validation function to check the value set
- Delegates to a notification function to notify interested parties of value changes

I can map these semantics directly to code, as shown in the following:

4. Mixins in this chapter are a cross between what's commonly known as a "protocol" and the Template method design pattern, minus the hierarchy.

```
var HoleMixin = {
  setValue: function(newValue) {
    var oldVal  = this._value;

    this.validate(newValue);
    this._value = newValue;
    this.notify(oldVal, newValue);
    return this._value;
  }
};
```

The `HoleMixin#setValue` method defines a set of circumstances that must be met in order for a type to qualify as a `Hole`. Any type extending `Hole` should offer `notify` and `validate` methods. However, there is no real `Hole` type yet, only a mixin that describes "holiness." The implementation of a `Hole` constructor is fairly simple:

```
var Hole = function(val) {
  Container.call(this, val);
}
```

The signature for the `Hole` constructor is the same as for `Container`; in fact, the use of the `Container.call` method taking the `Hole` instance's `this` pointer ensures that whatever `Container` does on construction will occur in the context of the `Hole` instance.

The mixin protocol specification for `HoleMixin` is as follows:

Extension protocol
: Must provide `notify`, `validate` and `init` methods

Interface protocol
: Constructor and `setValue`

The need for the `init` method is derived from the direct use of `Container` in the constructor. Failing to meet any given mixin, particularly the `Container` mixin, has potentially dire consequences:

```
var h = new Hole(42);
//TypeError: Object [object Object] has no method 'init'
```

That the `Container` extension interface was not met means that any attempt to use `Hole` at the moment will fail. But despair not; the interesting thing about mixin extension is that any given type is composed of existing mixins, either outright or through extension.

Based on the illustration shown in Figure 9-8, the fulfillment of the `Hole` type requires either implementing or mixing in both `ObserverMixin` and `ValidateMixin`.

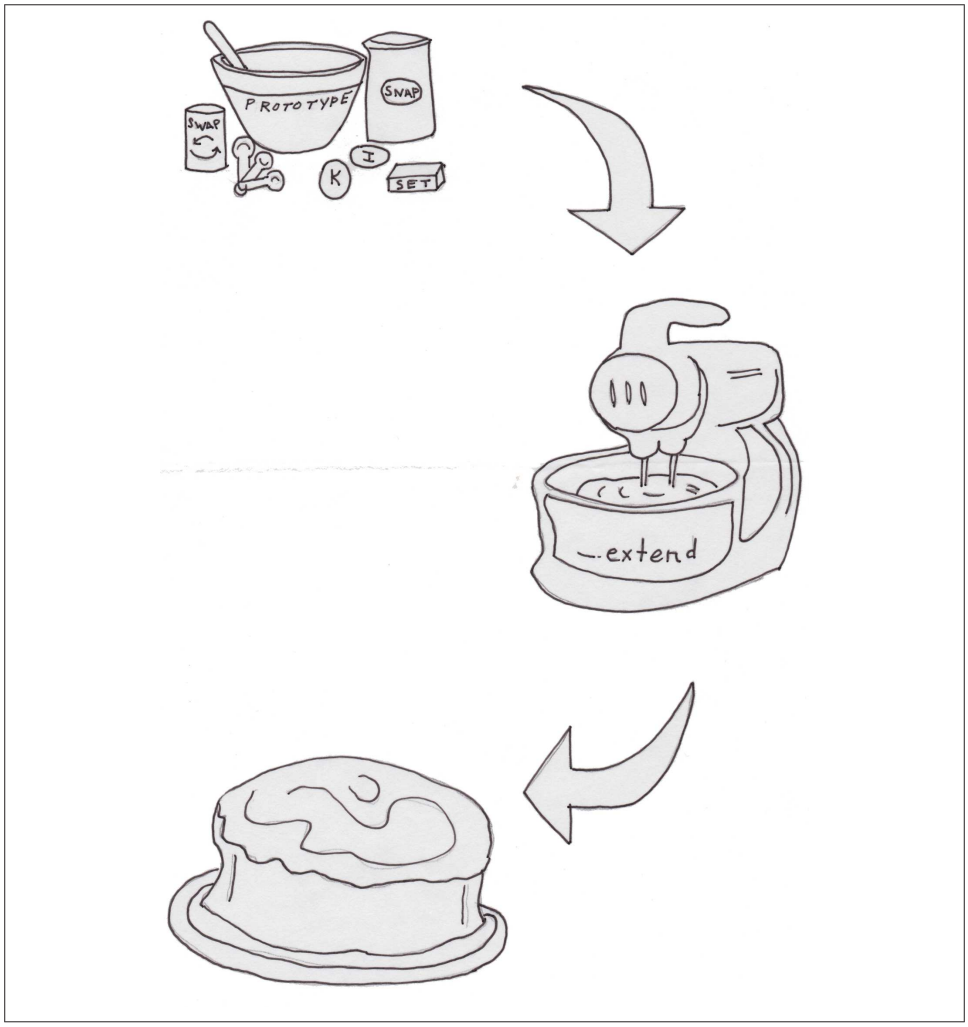

Figure 9-8. Using mixins to "mix" behaviors

Since neither of these mixins exist, I'll need to create them, starting with the `Observer Mixin`:

```
var ObserverMixin = (function() {
  var _watchers = [];

  return {
    watch: function(fun) {
      _watchers.push(fun);
      return _.size(_watchers);
    },
    notify: function(oldVal, newVal) {
```

```
      _.each(_watchers, function(watcher) {
        watcher.call(this, oldVal, newVal);
      });

      return _.size(_watchers);
    }
  };
}());
```

The use of the JavaScript closure mojo `(function() {...}());` to encapsulate the `_watchers` object is the common way to hide data, and it is therefore the preferred way to hide a bit of mixin state as well. The watch function takes a function of two values, the old value and the new, and adds it to the `_watchers` array. The `watch` method also returns the number of watchers stored. The `notify` method then loops over the `_watchers` and calls each function, finally returning the number of watchers notified. The implementation of `ObserverMixin` could be enhanced to be more robust in the face of watch function failure, and also to allow the removal of watchers, but I leave that as an exercise to the reader.[5]

The second missing mixin is the `ValidateMixin`, implemented as follows:

```
var ValidateMixin = {
  addValidator: function(fun) {
    this._validator = fun;
  },
  init: function(val) {
    this.validate(val);
  },
  validate: function(val) {
    if (existy(this._validator) &&
        !this._validator(val))
      fail("Attempted to set invalid value " + polyToString(val));
  }
};
```

As shown, it's the `ValidateMixin` that finally fulfills the `init` extension requirement. This makes sense since a valid initialization step is to validate the starting value of the container. The other two functions, `addValidator` and `validate`, set the validation function and call it (if set) respectively.

Now that the mixins are in place, it's time to mix them together to fulfill the requirements of the `Hole` type:

```
_.extend(Hole.prototype
         , HoleMixin
```

5. The ECMAScript.next effort has described an `Object.observe` method that works similarly to the features described herein. The specification should become a reality in JavaScript core sometime before the heat death of the sun. More information is found at *http://wiki.ecmascript.org/doku.php?id=harmony:observe*.

```
          , ValidateMixin
          , ObserverMixin);
```

I mentioned in Chapter 7 that Underscore's `_.extend` function is tricky because it modifies the target object. However, in the case of mixin extension, this behavior is exactly what I want. That is, by using `_.extend`, I copy all of the methods into `Hole.mixin`. So how does the fully mixed implementation work? Observe:

```
var h = new Hole(42);
```

That the constructor works at all is a good sign to start. What if I add a validator that is guaranteed to fail?

```
h.addValidator(always(false));

h.setValue(9);
// Error: Attempted to set invalid value 9
```

Since I attached a validator returning `false` in all cases, I'll never be able to set another value again unless I remove the validation function directly. However, let me create a new `Hole` instance with a less restrictive validator:

```
var h = new Hole(42);

h.addValidator(isEven);
```

The new instance should allow only even numbers as values:

```
h.setValue(9);
// Error: Attempted to set invalid value 9

h.setValue(108);
//=> 108

h;
//=> {_validator: function isEven(n) {...},
//    _value: 108}
```

That the `Hole` instance `h` allows only even numbers is of limited value, but for illustration purposes it serves well. Below I'll add a watcher using the `watch` method:

```
h.watch(function(old, nu) {
  note(["Changing", old, "to", nu].join(' '));
});
//=> 1

h.setValue(42);
// NOTE: Changing 108 to 42
//=> 42
```

Passing in the even number `42` shows that the watcher is called, so adding another should also work:

```
h.watch(function(old, nu) {
  note(["Veranderende", old, "tot", nu].join(' '));
});
//=> 2

h.setValue(36);
// NOTE: Changing 42 to 36
// NOTE: Veranderende 42 tot 36
//=> 36
```

So I've managed to create a new JavaScript type by both using constructor calls, and by mixing discrete packets of functionality into a coherent hole…I mean whole. In the next section, I'll talk about how to use mixin extension to add new capabilities to existing types.

New Semantics via Mixin Extension

Adding new capabilities to existing JavaScript types couldn't be simpler; you just muck with the `prototype` and, *kaboom* you've attached new behavior. Well, *kaboom* is the operative term here because it's rarely that simple. It's not always straightforward to extend existing types because you never know whether you might break some delicate internal balance. Keeping that in mind, I will explore how to extend the capabilities of the `Hole` type to include new change semantics. First, I like the idea of providing the `setValue` method as a low-level way to tap into the change machinery. However, I would like to provide another method, `swap`, that takes a function and some number of arguments and sets the new value based on the result of a call to said function with the current value and the arguments. The best way to present this idea is to show the implementation and some examples:

```
var SwapMixin = {
  swap: function(fun /* , args... */) {
    var args = _.rest(arguments)
    var newValue = fun.apply(this, construct(this._value, args));

    return this.setValue(newValue);
  }
};
```

The `swap` method on the `SwapMixin` indeed takes a function and some arguments. The new value is then calculated using the function given the `_value` and the additional arguments. The mixin protocol specification for `SwapMixin` is as follows:

Extension protocol
: Must provide a `setValue` method and a `_value` property

Interface protocol
: The `swap` method

I can actually test the `SwapMixin` in isolation:

```
var o = {_value: 0, setValue: _.identity};

_.extend(o, SwapMixin);

o.swap(construct, [1,2,3]);
//=> [0, 1, 2, 3]
```

So, as shown, the logic behind the `swap` mixin seems sound. Before I use it to enhance `Hole`, let me implement another mixin, `SnapshotMixin`, used to offer a way to safely grab the value in the `Hole` instance:

```
var SnapshotMixin = {
  snapshot: function() {
    return deepClone(this._value);
  }
};
```

The `SnapshotMixin` provides a new method named `snapshot` that clones any object contained therein. Now, the new specification of `Hole` stands as:

```
_.extend(Hole.prototype
         , HoleMixin
         , ValidateMixin
         , ObserverMixin
         , SwapMixin
         , SnapshotMixin);
```

And now, any new `Hole` instances will have the enhanced behavior:

```
var h = new Hole(42);

h.snapshot();
//=> 42

h.swap(always(99));
//=> 99

h.snapshot();
//=> 99
```

Mixin extension is not only a powerful way to define new types, but also useful for enhancing existing types. Bear in mind that there are caveats in that it's not always straightforward to extend existing types, and additionally any extension will take place globally.

New Types via Mixin Mixing

Now that I've shown how to define two base types (`Container` and `Hole`), let me implement one more called `CAS`, which offers compare-and-swap semantics. That is, any change to the type occurs based on an assumption that you know what the existing value happens to be. The definition starts by using the construction behavior of `Hole`:

```
var CAS = function(val) {
  Hole.call(this, val);
}
```

The interesting part of the definition of the `CASMixin` is that it overrides the `swap` method on the `SwapMixin` as shown here:

```
var CASMixin = {
  swap: function(oldVal, f) {
    if (this._value === oldVal) {
      this.setValue(f(this._value));
      return this._value;
    }
    else {
      return undefined;
    }
  }
};
```

The `CASMixin#swap` method takes two arguments instead of the one taken by `SwapMixin`. Additionally, the `CASMixin#swap` method returns `undefined` if the expected value does not match the actual `_value`. There are two ways to mix the implementation of the `CAS` types. First, I could simply leave out the `SwapMixin` on the extension and use the `CASMixin` instead, since I know that the `swap` method is the only replacement. However, I will instead use ordering to `_.extend` to take care of the override:

```
_.extend(CAS.prototype
         , HoleMixin
         , ValidateMixin
         , ObserverMixin
         , SwapMixin
         , CASMixin
         , SnapshotMixin);
```

While I knew that the `SwapMixin` was fully subsumed by the `CASMixin`, leaving it in is not entirely bad. The reason is that if I do not control the `SwapMixin`, then it's conceivable that it may gain enhancements at a future date beyond simply the `swap` method. By leaving in the extension chain, I get any enhancements for free in the future. If I do not like the future "enhancements," then I can choose to remove `SwapMixin` later. To wrap this section up, the `CAS` type is used as follows:

```
var c = new CAS(42);

c.swap(42, always(-1));
//=> -1

c.snapshot();
//=> -1

c.swap('not the value', always(100000));
//=> undefined
```

```
c.snapshot();
//=> -1
```

And that concludes the discussion of mixin extension. However, there is one more point to make about it: mixin extension, if done correctly, is an implementation detail. In fact, I would still reach for simple data like primitives, arrays, and objects (as maps) over mixin-based programming. Specifically, I've found that when you're dealing with a large number of data elements, then simple data is best because you can use common tools and functions to process it—the more generic data processing tools available, the better. On the other hand, you will definitely find a need to create highly specialized types with well-defined interfaces driving per-type semantics.[6] It's in the case of these specialized types that I've found mixin-based development a real advantage.

Simple data is best. Specialized data types should be, well, special.

Methods Are Low-Level Operations

That the types created in the previous sections are object/method-centric is a technical detail that need not leak into a functional API. As I've stressed throughout this book, functional APIs are composable and if created well, do not require explicit knowledge of the intermediate types between composition points. Therefore, by simply creating a function-based API for accessing and manipulating the container types, I can hide most of the detail of their implementation.

First, let me start with the container:

```
function contain(value) {
  return new Container(value);
}
```

Simple, right? If I were providing a container library, then I would offer the `contain` function as the user-facing API:

```
contain(42);
//=> {_value: 42} (of type Container, but who cares?)
```

For developers, I might additionally provide the mixin definitions for extension purposes.

The `Hole` functional API is similar, but beefier:

```
function hole(val /*, validator */) {
  var h = new Hole();
  var v = _.toArray(arguments)[1];
```

6. If you come from a Scala background, then the mixin-based development outlined here is far from realizing the well-known Cake pattern (Wampler 2009). However, with some work and runtime mixin inspection, you can achieve a rough approximation, providing an additional capability for large-scale module definition.

```
  if (v) h.addValidator(v);

  h.setValue(val);

  return h;
}
```

I've managed to encapsulate a lot of the logic of validation within the confines of the `hole` function. This is ideal because I can compose the underlying methods in any way that I want. The usage contract of the `hole` function is much simpler than the combined use of the `Hole` constructor and the `addValidator` method:

```
var x = hole(42, always(false));
// Error: Attempted to set invalid value 42
```

Likewise, although `setValue` is a method on the type, there is no reason to expose it functionally. Instead, I can expose just the `swap` and `snapshot` functions instead:

```
var swap = invoker('swap', Hole.prototype.swap);
```

And the `swap` function works as any `invoker`-bound method, with the target object as the first argument:

```
var x = hole(42);

swap(x, sqr);
//=> 1764
```

Exposing the functionality of the `CAS` type is very similar to `Hole`:

```
function cas(val /*, args */) {
  var h = hole.apply(this, arguments);
  var c = new CAS(val);
  c._validator = h._validator;

  return c;
}

var compareAndSwap = invoker('swap', CAS.prototype.swap);
```

I'm using (abusing) private details of the `Hole` type to implement most of the capability of the `cas` function, but since I control the code to both types, I can justify the coupling. In general, I would avoid that, especially if the abused type is not under my immediate control.

Finally, I can now implement the remaining container functions as generic delegates:

```
function snapshot(o) { return o.snapshot() }
function addWatcher(o, fun) { o.watch(fun) }
```

And these functions work exactly how you might guess:

```
var x = hole(42);

addWatcher(x, note);

swap(x, sqr);
// NOTE: 42 chapter01.js:38
//=> 1764

var y = cas(9, isOdd);

compareAndSwap(y, 9, always(1));
//=> 1

snapshot(y);
//=> 1
```

I believe that by putting a functional face on the container types, I've achieved a level of flexibility not obtainable via an object/method focus.

}).call("Finis");

This chapter concludes my coverage of functional programming in JavaScript by showing how it lends to building software. Even though some problems seemingly call for object or class-based thinking, very often there are functional ways to achieve the same goals. Not only will there be functional ways to build parts of your system, but building your system functionally often leads to more flexibility in the long term by not tying your users to an object-centric API.

Likewise, even when a problem calls for object-thinking, approaching the problem with a functional eye can lead to vastly different solutions than object-oriented programming dictates. If functional composition has proven useful, how might object composition fare? In this chapter, I discussed mixin-based design and how it is indeed a functionally flavored style of object composition.

Writing this book has been a joy for me and I hope has been an enlightening adventure for you. Learning functional programming shouldn't be seen as a goal in itself, but instead a technique for achieving your goals. There may be times when it is just not a good fit, but even then, thinking functionally can and will help change the way you build software in general.