Luis Atencio

# Functional Programming in JavaScript

How to improve your JavaScript programs using functional techniques

*SAMPLE CHAPTER*

*Functional Programming in JavaScript*

by Luis Atencio

**Chapter 5**

Copyright 2016 Manning Publications

# *brief contents*

# 5 Design patterns against complexity

***This chapter covers***

- The issues with imperative error-handling schemes
- Using containers to prevent access to invalid data
- Implementing functors as a mechanism for data transformation
- Understanding monads as data types that facilitate composition
- Consolidating error-handling strategies with monadic types
- Interleaving and composing monadic types

> Null-references … was a billion-dollar mistake.
>
> —Tony Hoare, InfoQ

Some people mistakenly view functional programming as a paradigm devoted only to academic problems, mostly numerical in nature, that are, for the most part, oblivious to the probabilities of failure real-life systems deal with. In recent years,

however, people are finding that functional programming can treat error handling more elegantly than any other development style.

Many issues can arise in software where data inadvertently becomes `null` or `undefined`, exceptions are thrown, or network connectivity is lost, to name a few. Our code needs to account for the potential of any of these issues occurring, which unavoidably creates complexity. As a result, we spend countless hours making sure our code throws and catches the proper exceptions and checks for `null` values everywhere we can think of, and what do we get? Even more complex code—code that doesn't scale and becomes harder to reason about as the size and complexity of applications increase.

We need to work smarter, not harder. In this chapter, I'll introduce the concept of *functors* as a means to create simple data types on which functions can be mapped. A functor is applied to data types called *monads* that contain specific behavior for dealing with errors in different ways. Monads are one of the hardest concepts to grasp in functional programming because the theory is deeply rooted in category theory, which I won't cover. My intention is to focus only on the practical aspects. Having said that, I'll slowly work my way into that topic, layering in some prerequisite concepts, and then show how you can use monads to create fault-tolerant function compositions in a way that imperative error-handling mechanism can't.

## 5.1 Shortfalls of imperative error handling

JavaScript errors can occur in many situations, especially when an application fails to communicate with a server or tries to access properties of a `null` object. Also, third-party libraries can have functions throw exceptions to signal special error conditions. Hence, we always need to be prepared for the worst and design with failure in mind, instead of letting it become an afterthought and regretting it later. In the imperative world, exceptions are handled via the `try-catch` idiom.

### 5.1.1 Error handling with try-catch

JavaScript's current exception-handling mechanism is geared toward throwing and catching exceptions through the popular `try-catch` structure present in most modern programming languages:

```
try {
   // code that might throw an exception in here
}
catch (e) {
   // statements to handle any exceptions
   console.log('ERROR' + e.message);
}
```

The purpose of this structure is to surround a piece of code that you deem to be unsafe. Upon throwing an exception, the JavaScript runtime abruptly halts the

program's execution and creates a stack trace of all function calls leading up to the problematic instruction. As you know, specific details about the error, such as the message, line number, and filename, are populated into an object of type `Error` and passed into the `catch` block. The `catch` block becomes a safe haven so that you can potentially recover your program. For example, recall the `findObject` and `findStudent` functions:

```
// findObject :: DB, String -> Object
const findObject = R.curry(function (db, id) {
  const result = find(db, id)
  if(!result) {
     throw new Error('Object with ID [' + id + '] not found');
  }
  return result;
});

// findStudent :: String -> Student
const findStudent = findObject(DB('students'));
```

Because any of these functions can throw an exception, in practice you would need to enclose them in a `try-catch` block when calling them:

```
try {
    var student = findStudent('444-44-4444');
}
catch (e) {
    console.log('ERROR' + e.message);
}
```

Just as you abstracted loops and conditional statements with functions before, you need to abstract error handling. Clearly, functions that use `try-catch` as shown here can't be composed or chain together and put a great deal of pressure on the design of your code.

### 5.1.2 Reasons not to throw exceptions in functional programs

The structured mechanism of throwing and catching exceptions in imperative JavaScript code has many drawbacks and is incompatible with the functional design. Functions that throw exceptions

- Can't be composed or chained like other functional artifacts.
- Violate the principle of referential transparency that advocates a single, predictable value, because throwing exceptions constitutes another exit path from your function calls.
- Cause side effects to occur because an unanticipated unwinding of the stack impacts the entire system beyond the function call.

- Violate the principle of non-locality because the code used to recover from the error is distanced from the originating function call. When an error is thrown, a function leaves the local stack and environment:

```
try {
     var student = findStudent('444-44-4444');

     ... more lines of code in between
}
catch (e) {
     console.log('ERROR: not found');

     // Handle error here
}
```

- Put a great deal of responsibility on the caller to declare matching `catch` blocks to manage specific exceptions instead of just worrying about a function's single return value.
- Are hard to use when multiple error conditions create nested levels of exception-handling blocks:

```
var student = null;
try {
   student = findStudent('444-44-44444');
}
catch (e) {
   console.log('ERROR: Cannot locate students by SSN');

   try {
      student = findStudentByAddress(new Address(...));
   }
   catch (e) {
      console.log('ERROR: Student is no where to be found!');
   }
}
```

You're probably asking yourself, "Is throwing exceptions completely off the table in functional programming?" I don't believe so. In practice, they can never be off the table, because there are many factors outside of your control that you need to account for. Also, you may be writing code against a library outside of your control that implements exceptions.

Using exceptions can be effective for certain edge cases. In `checkType` in chapter 4, you used an exception to signal a fundamental misuse of the API. They're also useful to signal unrecoverable conditions like `RangeError: Maximum call stack size exceeded`, which I'll talk about in chapter 7. Throwing exceptions has a place but shouldn't be done excessively. A common scenario that occurs in JavaScript is the infamous `TypeError` resulting from invoking a function on a `null` object.

### 5.1.3 Problems with null-checking

The alternative to failing abruptly from a function call is to return `null`. That, at least, guarantees only one route that leaves a function call, but it's not any better. Functions that return `null` create a different responsibility for users: pesky `null` checks. Consider the function `getCountry`, which is in charge of reading a student's address and then country:

```
function getCountry(student) {
   let school = student.getSchool();
   if(school !== null) {
     let addr = school.getAddress();
     if(addr !== null) {
        var country = addr.getCountry();
        return country;
     }
     return null;
   }
   throw new Error('Error extracting country info');
}
```

At a glance, this function should have been simple to implement—after all, it's just extracting an object's property. I could have created a simple lens that focuses on this property; in the event of a `null` address, a lens is smart enough to return `undefined`, but it doesn't help me to print an error message.

Instead, I ended up with lots of lines of code to defend myself from unexpected behavior. Defensively wrapping code with lots of `try-catch` or `null` checks is cowardly. Wouldn't it be great to be able to handle errors effectively while avoiding all of this unnecessary boilerplate code?

## 5.2 Building a better solution: functors

Functional error handling is a radically different approach to properly cope with the adversities found in software systems. The idea, however, is somewhat similar: create a safety box (a container, if you will) around potentially hazardous code (see figure 5.1).

```
try {
    var student = findStudent('444-44-4444');

    ... more lines of code
}
catch (e) {
     console.log('ERROR: Student not found!');

     // Handle missing student
}
```

**Figure 5.1 The `try-catch` structure invisibly creates a safety box around functions that can throw exceptions. This safety box is materialized into a container.**

In functional programming, this notion of boxing the dangerous code still applies, but you throw away the `try-catch` block. Now, here's the big difference. Walling off impurity is made a first-class citizen in functional programming by the use of functional data types. Let's begin with the most basic type and move into the more advanced ones.

### 5.2.1 Wrapping unsafe values

Containerizing (or wrapping) values is a fundamental design pattern in functional programming because it guards direct access to the values so they can be manipulated safely and immutably in your programs. It's like wearing armor before going to battle. Accessing a wrapped value can only be done by *mapping an operation to its container*. In this chapter, I'll talk extensively about the concept of a *map*, but you already learned about this in chapter 3 when you used `map` on arrays—the array was the container of values, in that case.

It turns out you can map functions to much more than just arrays. In functional JavaScript, *a map is nothing more than a function*; this idea comes from referential transparency, where a function must always "map to" the same result given the same input. So you can also think of map as a gate that allows you to plug in a lambda expression with specific behavior that transforms an encapsulated value. In the case of arrays, you used `map` to create a new array with the transformed values.

Let's illustrate this concept with a simple data type called `Wrapper`, in the following listing. Although this type is simple, the underlying principle is extremely powerful and will pave the way for the next sections in this chapter, so it's important that you understand it.

**Listing 5.1 Functional data type to wrap values**

```
class Wrapper {
    constructor(value) {
        this._value = value;
    }

    // map :: (A -> B) -> A -> B
    map(f) {
      return f(this._value);
    };

    toString() {
      return 'Wrapper (' + this._value + ')';
    }
}

// wrap :: A -> Wrapper(A)
const wrap = (val) => new Wrapper(val);
```

**Simple type that stores a single value of any type**

**Maps a function over this type (just like arrays)**

**Helper function that quickly creates wrappers around values**

You can use a wrapper object to encapsulate a potentially erroneous value. Because you won't have direct access to it, the only way to extract it is to use the `identity`

function you learned about in chapter 4 (notice there's no explicit `get` method on this wrapper type). Certainly JavaScript will give you easy access to this value, but the point to understand here is that once the value enters the container, it can't be directly retrieved or transformed (like a virtual barrier); see figure 5.2.

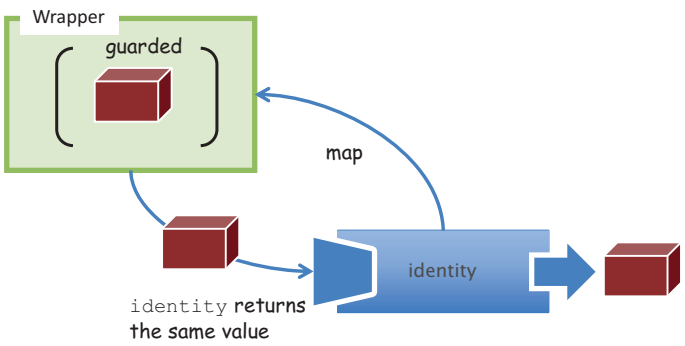


**Figure 5.2 The `Wrapper` type uses `map` to safely access and manipulate values. In this case, you're mapping the identity function over the container to extract the value as is from the container.**

Here's a concrete example using a valid value:

```
const wrappedValue = wrap('Get Functional');
wrappedValue.map(R.identity); //-> 'Get Functional'
```

**Extracts the value**

You can map any function to this container to either log to the console or manipulate it as needed:

```
wrappedValue.map(console.log);
wrappedValue.map(R.toUpper); //-> 'GET FUNCTIONAL'
```

**Runs the function over the internal value**

The benefit of this simple idea is that any code written against these wrappers needs to be able to "reach into the container" via `Wrapper.map` in order to use the guarded value contained within. But if the value happens to be `null` or `undefined`, the responsibility is placed on the caller, which may or may not gracefully handle this case. Later, you'll see a better alternative:

```
const wrappedNull = wrap(null);
wrappedNull.map(doWork);
```

**doWork is given the burden of null-checking.**

As you can see from this example, to manipulate a value within a guarded, wrapped context, you need to apply a function to it; you can't invoke a function directly. What

to do in the event of an error can be delegated to concrete wrapper types. In other words, you can check for `null` before calling the function, or check for an empty string, a negative number, and so on. Hence, the semantic of `Wrapper.map` is determined by the specific implementation of the wrapping type.

Let's not get ahead of ourselves; we have some more groundwork to cover. Consider this slightly different variation of `map`, called `fmap`:

```
// fmap :: (A -> B) -> Wrapper[A] -> Wrapper[B]
fmap (f) {
  return new Wrapper(f(this._value));
}
```

**Wraps the transformed value in the container before returning it to the caller**

`fmap` knows how to apply functions to values wrapped in a context. It first opens the container, then applies the given function to its value, and finally closes the value back into a new container of the same type. This type of function is known as a *functor*.

### 5.2.2 Functors explained

In essence, a functor is nothing more than a data structure that you can map functions over with the purpose of lifting values into a wrapper, modifying them, and then putting them back into a wrapper. It's a design pattern that defines semantics for how `fmap` should work. Here's the general definition of `fmap`:

```
fmap :: (A -> B) -> Wrapper(A) -> Wrapper(B)
```

**Wrapper is any container type.**

The function `fmap` takes a function (from `A -> B`) and a functor (wrapped context) `Wrapper(A)` and returns a new functor `Wrapper(B)` containing the result of applying said function to the value and closing it once more. Figure 5.3 shows a quick example that uses the `increment` function as a mapping function from `A -> B` (except in this case, `A` and `B` are the same types).

Notice that because `fmap` basically returns a new copy of the container at each invocation, much as lenses (chapter 2) work, it can be considered immutable. In figure 5.3, mapping the `increment` over `Wrapper(1)` returns a completely new object,

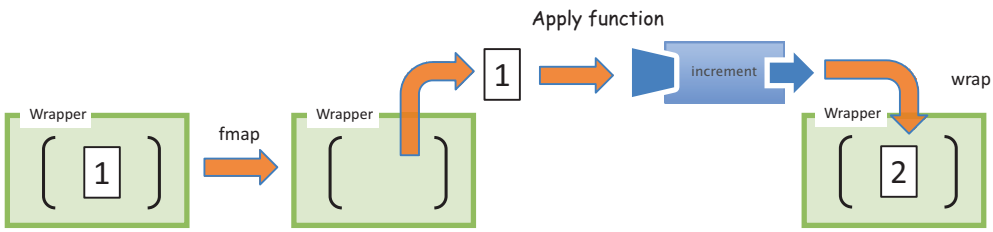


**Figure 5.3 A value of 1 is contained within `Wrapper`. The functor is called with the wrapper and the `increment` function, which transforms the value internally and closes it back into a container.**

`Wrapper(2)`. Let's go over a simple example before you begin applying functors to solve more-practical problems. Consider a simple `2 + 3 = 5` addition using functors. You can curry an `add` function to create a `plus3` function:

```
const plus = R.curry((a, b) => a + b);
const plus3 = plus(3);
```

Now you'll store the number 2 into a `Wrapper` functor:

```
const two = wrap(2);
```

Calling `fmap` to map `plus3` over the container performs addition:

```
const five = two.fmap(plus3); //-> Wrapper(5)
five.map(R.identity); //-> 5
```

**Returns the value in a context**

The outcome of `fmap` yields another context of the same type, which you can map `R.identity` over to extract its value. Notice that because the value never escapes the wrapper, you can map as many functions as you want to it and transform its value at each step of the way:

```
two.fmap(plus3).fmap(plus10); //-> Wrapper(15)
```

This can be a bit tricky to understand, so figure 5.4 shows how `fmap` works with `plus3`.

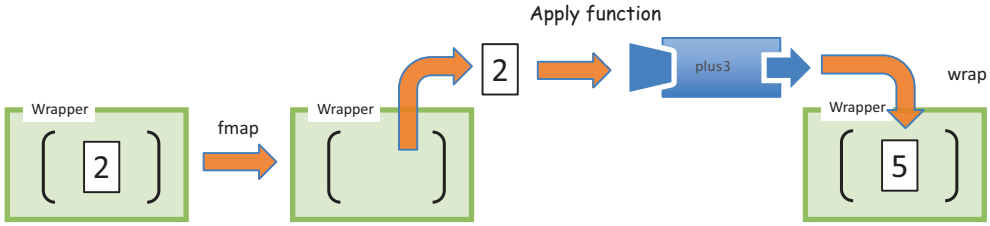


**Figure 5.4 The value 2 has been added to a `Wrapper` container. The functor is used to manipulate this value by unwrapping it from the context, applying the given function to it, and rewrapping the value back into a new context.**

The purpose of having `fmap` return the same type (or wrap the result again into a container of the same type) is so you can continue chaining operations. Consider the following example, which maps `plus3` on a wrapped value and logs the result.

**Listing 5.2 Chaining functors to apply additional behavior to a given context**

```
const two = wrap(2);
two.fmap(plus3).fmap(R.tap(infoLogger)); //-> Wrapper(5)
```

Running this code prints the following message on the console:

```
InfoLogger [INFO] 5
```

Does this pattern of chaining functions look familiar? This is intentional: you've been using functors all along without realizing it. This is exactly what the `map` and `filter` functions do for arrays (you can review sections 3.3.2 and 3.3.4 if you need to):

```
map    :: (A -> B)       -> Array(A) -> Array(B)
filter :: (A -> Boolean) -> Array(A) -> Array(A)
```

`map` and `filter` are type-preserving functors, which is what activates the chaining pattern. Consider another functor you've seen all along: `compose`. As you learned in chapter 4, it's a mapping from functions into other functions (also type-preserving):

```
compose :: (B -> C) -> (A -> B) -> (A -> C)
```

Functors, like any other functional programming artifact, are governed by some important properties:

- *They must be side effect–free.* You can map the `R.identity` function to obtain the same value over a context. This proves functors are side effect–free and preserves the structure of the wrapped value:

  ```
  wrap('Get Functional').fmap(R.identity); //-> Wrapper('Get Functional')
  ```

- *They must be composable.* This property indicates that the composition of a function applied to `fmap` should be exactly the same as chaining `fmap` functions together. As a result, the following expression is exactly equivalent to the program in listing 5.2:

  ```
  two.fmap(R.compose(plus3, R.tap(infoLogger))).map(R.identity); //-> 5
  ```

It's no surprise that functors have these requirements. As a result, they're prohibited from throwing exceptions, mutating elements, or altering a function's behavior. Their practical purpose is to create a context or an abstraction that allows you to securely manipulate and apply operations to values without changing any original values. This is evident in the way `map` transforms one array into another without altering the original array; this concept equally translates to any container type.

But functors by themselves aren't compelling, because they're not expected to know how to handle cases with `null` data. Ramda's `R.compose`, for instance, will break if a `null` function reference is passed into it. This isn't a flaw in the design; it's intentional. *Functors map functions of one type to another.* More-specialized behavior can be found in functional data types called *monads*. Among other things, monads can streamline error handling in your code, allowing you to write fluent function compositions. What's their relationship to functors? Monads are the containers that functors "reach into."

Don't let the term *monad* discourage you; if you've written jQuery code, then monads should be familiar. Behind all the complicated rules and theories, the purpose of monads is to provide an abstraction over some resource—whether it's a simple value, a DOM element, an event, or an AJAX call—so that you can safely process the data contained within it. In this respect, you can classify jQuery as a DOM monad:

```
$('#student-info').fadeIn(3000).text(student.fullname());
```

This code behaves like a monad because jQuery is taking charge of applying the `fadeIn` and `text` transformations safely. If the `student-info` panel doesn't exist, applying methods to the empty jQuery object will fail gracefully rather than throw exceptions. Monads aimed at error handling have this powerful quality: safely propagating errors so your application is fault-tolerant. Let's dive into monads next.

## 5.3 Functional error handling using monads

Monads solve all the problems of traditional error handling outlined earlier when applied to functional programs. But before diving into this topic, let's first understand a limitation in the use of functors. As you saw earlier, you can use functors to safely apply functions to values in an immutable and safe manner. But when used throughout your code, functors can easily get you into an uncomfortable situation. Consider an example of fetching a student record by SSN and then extracting its address property. For this task, you can identify two functions—`findStudent` and `getAddress`—both using functor objects to create a safe context around their returned values:

```
const findStudent = R.curry((db, ssn) =>
    wrap(find(db, ssn));
);

const getAddress = student =>
  wrap(student.fmap(R.prop('address')));
```

**Wraps the fetched object to safeguard against the possibility of not finding an object**

**Maps Ramda's R.prop() function over the object to extract its address, and then wraps the result**

Just as you've done all along, to run this program, you compose both functions together:

```
const studentAddress = R.compose(
    getAddress,
    findStudent(DB('student'))
);
```

Although you avoid all error-handling code, the result isn't what you expect. Instead of a wrapped address object, the returned value is a doubly wrapped address object:

```
studentAddress('444-44-4444'); //-> Wrapper(Wrapper(address))
```

In order to extract this value, you have to apply `R.identity` twice:

```
studentAddress('444-44-4444').map(R.identity).map(R.identity);
```

**Ugh!**

Certainly you don't want to access data this way in your code; just think about the case when you have three or four composed functions. You need a better solution. Enter monads.

### 5.3.1 Monads: from control flow to data flow

Monads are similar to functors, except that they can delegate to special logic when handling certain cases. Let's examine this idea with a quick example. Consider applying a function `half :: Number -> Number` to any wrapped value, as shown in figure 5.5:

```
Wrapper(2).fmap(half); //-> Wrapper(1)
Wrapper(3).fmap(half); //-> Wrapper(1.5)
```

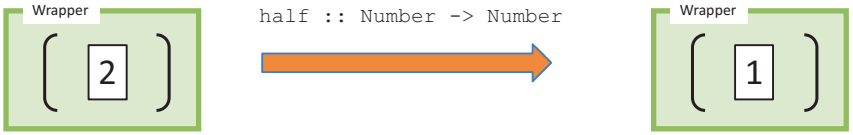


**Figure 5.5 Functors apply a function to a wrapped value. In this case, the wrapped value 2 is halved, returning a wrapped value of 1.**

But now suppose you want to restrict `half` to even numbers only. As is, the functor only knows how to apply the given function and close the result back in a wrapper; it has no additional logic. What can you do if you encounter an odd input value? You could return `null` or throw an exception. But a better strategy is to make this function more honest about how it handles each case and state that it returns a valid number when given the correct input value, or ignores it otherwise.

In the spirit of `Wrapper`, consider another container called `Empty`:

```
class Empty
  map(f) {
    return this;
  }

  toString() {
    return 'Empty ()';
  }
};

const empty = () => new Empty();
```

**noop. Empty doesn't store a value; it represents the concept of "empty" or "nothing."**

**Similarly, mapping a function to an Empty skips the operation.**

With this new requirement, you can implement `half` in the following way (figure 5.6):

```
const isEven = (n) => Number.isFinite(n) && (n % 2 == 0);
const half = (val) => isEven(val) ? wrap(val / 2) : empty();

half(4); //-> Wrapper(2)
half(3); //-> Empty
```

**Helper function distinguishes between odd and even numbers**

**Function half only works on even numbers, returning an empty container otherwise**

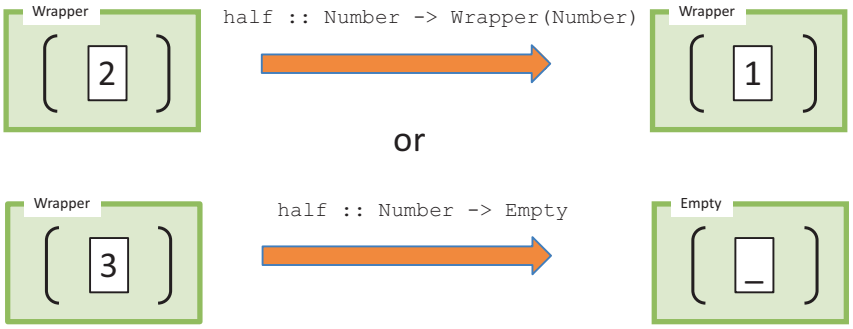


**Figure 5.6 Function `half` can return either a wrapped value or an empty container, depending on the nature of the input.**

A monad exists when you create a whole data type around this idea of lifting values inside containers and defining the rules of containment. Like functors, it's a design pattern used to describe computations as a sequence of steps without having any knowledge of the value they're operating on. Functors allow you to protect values, but when used with composition, monads are what let you manage data flow in a safe and side effect–free manner. In the previous example, you return an `Empty` container instead of `null` when trying to halve an odd number, which lets you apply operations on values without being concerned about errors that occur:

```
half(4).fmap(plus3); //-> Wrapper(5)
half(3).fmap(plus3); //-> Empty
```

**The implicit container knows how to map functions even when input is invalid.**

Monads can be targeted at a variety of problems. The ones we'll study in this chapter can be used to consolidate and control the complexity of imperative error-handling mechanisms and, thus, allow you to reason about your code more effectively.

Theoretically, monads are dependent on the type system of a language. In fact, many people advocate that you can only understand them if you have explicit types, as in Haskell. But you'll see that having a typeless language like JavaScript makes monads easy to read and frees you from having to deal with all the intricacies of a static type system.

You need to understand these two important concepts:

- *Monad*—Provides the abstract interface for monadic operations
- *Monadic type*—A particular concrete implementation of this interface

Monadic types share a lot of the same principles as the `Wrapper` object you learned about at the beginning of the chapter. But every monad is different and, depending on its purpose, can define different semantics driving its behavior (that is, for how `map` or `fmap` should work). These types define what it means to chain operations or nest functions of that type together, yet all must abide by the following interface:

- *Type constructor*—Creates monadic types (similar to the `Wrapper` constructor).
- *Unit function*—Inserts a value of a certain type into a monadic structure (similar to the `wrap` and `empty` functions you saw earlier). When implemented in the monad, though, this function is called `of`.
- *Bind function*—Chains operations together (this is a functor's `fmap`, also known as `flatMap`). From here on, I'll use the name `map`, for short. By the way, this bind function has nothing to do with the function-binding concept of chapter 4.
- *Join operation*—Flattens layers of monadic structures into one. This is especially important when you're composing multiple monad-returning functions.

Applying this new interface to the `Wrapper` type, you can refactor it in the following way.

**Listing 5.3 `Wrapper` monad**

```
class Wrapper {                                  <-- Type constructor
  constructor(value) {
    this._value = value;
  }

  static of(a) {                                 <-- Unit function
    return new Wrapper(a);
  }

  map(f) {                                       <-- Bind function (the functor)
    return Wrapper.of(f(this._value));
  }

  join() {                                       <-- Flattens nested layers
    if(!(this._value instanceof Wrapper)) {
      return this;
    }
    return this._value.join();
  }

  get() {
    return this._value;
  }

  toString() {                                   <-- Returns a textual representation of this structure
    return `Wrapper (${this._value})`;
  }
}
```

`Wrapper` uses the functor `map` to lift data into the container so that you can manipulate it side effect–free—walled off from the outside world. Not surprisingly, the `_.identity` function is used to inspect its contents:

```
Wrapper.of('Hello Monads!')
   .map(R.toUpper)
   .map(R.identity); //-> Wrapper('HELLO MONADS!')
```

The `map` operation is considered a *neutral functor* because it does nothing more than map the function and close it. Later, you'll see other monads add their own special touches to `map`. The `join` function is used to flatten nested structures—like peeling an onion. This can be used to eliminate the issues found with functors earlier, as shown next.

**Listing 5.4 Flattening a monadic structure**

```
// findObject :: DB -> String -> Wrapper
const findObject = R.curry((db, id) => {
   return Wrapper.of(find(db, id));
});

// getAddress :: Student -> Wrapper
const getAddress = student => {
   return Wrapper.of(student.map(R.prop('address')));
};

const studentAddress = R.compose(getAddress, findObject(DB('student')));

studentAddress('444-44-4444').join().get(); // Address
```

Because the composition in listing 5.4 returns a set of nested wrappers, the `join` operation is used to flatten out the structure into a single layer, as in this example:

```
Wrapper.of(Wrapper.of(Wrapper.of('Get Functional'))).join();

//-> Wrapper('Get Functional')
```

Figure 5.7 illustrates the join operation.

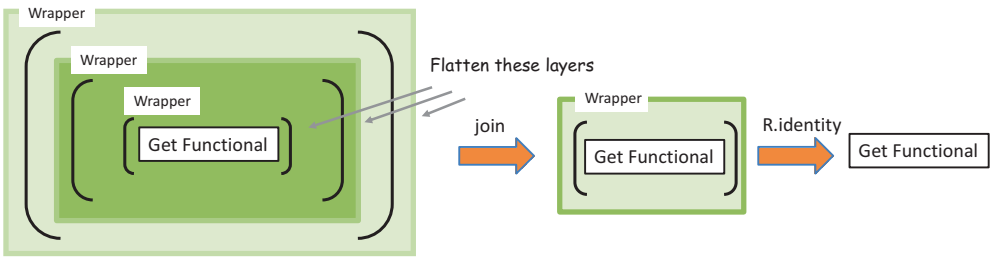


**Figure 5.7 Using the `join` operation to recursively flatten a nested monad structure, like peeling an onion**

With regard to arrays (which are also containers that can be mapped to), this is analogous to the `R.flatten` operation:

```
R.flatten([1, 2, [3, 4], 5, [6, [7, 8, [9, [10, 11], 12]]]]);

//=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

Monads typically have many more operations that support their specific behavior, and this minimal interface is merely a subset of its entire API. A monad itself, though, is abstract and lacks any real meaning. Only when implemented as a concrete type does its power begins to shine. Fortunately, most functional programming code can be implemented with just a few popular concrete types, which eliminates lots of boilerplate code while achieving an immense amount of work. Now, let's look at some full-fledged monads: `Maybe`, `Either`, and `IO`.

### 5.3.2 Error handling with Maybe and Either monads

In addition to wrapping valid values, monadic structures can also be used to model the absence of one—as `null` or `undefined`. Functional programming *reifies* errors (turns them into a "thing") by using the `Maybe` and `Either` types to do the following:

- Wall off impurity
- Consolidate `null`-check logic
- Avoid exception throwing
- Support compositionally of functions
- Centralize logic for providing default values

Both types provide these benefits in their own way. I'll begin with the `Maybe` monad.

#### CONSOLIDATING NULL CHECKS WITH MAYBE

The `Maybe` monad focuses on effectively consolidating `null`-check logic. `Maybe` is an empty type (a marker type) with two concrete subtypes:

- `Just(value)`—Represents a container that wraps a defined value.
- `Nothing()`—Represents either a container that has no value or a failure that needs no additional information. In the case of a `Nothing`, you can still apply functions over its (in this case, nonexistent) value.

These subtypes implement all the monadic properties you saw earlier, as well as some additional behavior unique to their purpose. Here's an implementation of `Maybe`.

**Listing 5.5 `Maybe` monad with subclasses `Just` and `Nothing`**

```
class Maybe {
  static just(a) {
     return new Just(a);
  }

  static nothing() {
     return new Nothing();
  }
```

**Container type (parent class)**

```
  static fromNullable(a) {
     return a !== null ? Maybe.just(a) : Maybe.nothing();
  }

  static of(a) {
     return just(a);
  }

  get isNothing() {
     return false;
  }

  get isJust() {
     return false;
  }
}

class Just extends Maybe {
   constructor(value) {
      super();
      this._value = value;
   }

   get value() {
      return this._value;
   }

   map(f) {
      return Maybe.fromNullable(f(this._value));
   }

   getOrElse() {
      return this._value;
   }

   filter(f) {
      Maybe.fromNullable(f(this._value) ? this._value : null);
   }

   chain(f) {
      return f(this._value);
   }

   toString () {
     return `Maybe.Just(${this._value})`;
   }
}

class Nothing extends Maybe {

   map(f) {
    return this;
   }

   get value()  {
     throw new TypeError("Can't extract the value
        of a Nothing.");
   }
```

**Builds a Maybe from a nullable type (constructor function). If the value lifted in the monad is null, instantiates a Nothing; otherwise, stores the value in a Just subtype to handle the presence of a value.**

**Subtype Just to handle the presence of a value**

**Maps a function to Just, transforms its value, and stores it back into the container**

**Extracts the value from the structure or a provided default monad unity operation**

**Returns a textual representation of this structure**

**Subtype Nothing to protect against the absence of a value**

**Attempting to extract a value from a Nothing type generates an exception indicating a bad use of the monad (I'll discuss this shortly); otherwise, the value is returned.**

```
  getOrElse(other) {
    return other;
  }

  filter(f) {
    return this._value;
  }

  chain(f) {
     return this;
  }

  toString() {
   return 'Maybe.Nothing';
  }
}
```

**Ignores the value and returns the other**

**If a value is present and matches the given predicate, returns a Just describing the value; otherwise returns a Nothing**

**Returns a textual representation of this structure**

`Maybe` explicitly abstracts working with "nullable" values (`null` and `undefined`) so you're free to worry about more important things. As you can see, `Maybe` is basically an abstract umbrella object for the concrete monadic structures `Just` and `Nothing`, each containing its own implementations of the monadic properties. I mentioned earlier that the implementation for the behavior of the monadic operations ultimately depends on the semantics imparted by a concrete type. For instance, `map` behaves differently depending on whether the type is a `Nothing` or a `Just`. Visually, a `Maybe` structure can store a student object as shown in figure 5.8:

```
// findStudent :: String -> Maybe(Student)
function findStudent(ssn)
```

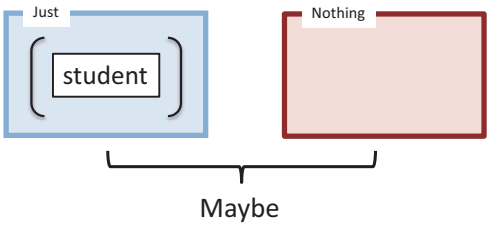


**Figure 5.8 A `Maybe` structure has two subtypes: `Just` and `Nothing`. Calling `findStudent` returns its value wrapped in `Just` or the absence of a value in `Nothing`.**

This monad is frequently used with calls that contain uncertainty: querying a database, looking up values in a collection, requesting data from the server, and so on. Let's continue with the example started in listing 5.4 of extracting the `address` property of a student object that's fetched from a local store. Because a record may or may not exist, you wrap the result of the fetch in a `Maybe` and add the `safe` prefix to these operations:

```
// safeFindObject :: DB -> String -> Maybe
const safeFindObject = R.curry((db, id) => {
   return Maybe.fromNullable(find(db, id));
});
```

```
// safeFindStudent :: String -> Maybe(Student)
const safeFindStudent = safeFindObject(DB('student'));

const address = safeFindStudent('444-44-4444').map(R.prop('address'));
address; //-> Just(Address(...)) or Nothing
```

Another benefit of wrapping results with monads is that it embellishes the function signature, making it self-documented and honest about the uncertainty of its return value. `Maybe.fromNullable` is useful because it handles the `null`-checking on your behalf. Calling `safeFindStudent` will produce a `Just(Address(...))` if it encounters a valid value or a `Nothing` otherwise. Mapping `R.prop` over the monad behaves as expected. In addition, it does a good job of detecting programmatic errors or misuses of an API call: you can use it to enforce preconditions indicating whether parameters are permitted to be invalid. If an invalid value is passed into `Maybe.fromNullable`, it produces a `Nothing` type, such that calling `get()` to open the container will throw an exception:

```
TypeError: Can't extract the value of a Nothing.
```

Monads expect you to stick to mapping functions over them instead of directly extracting their contents. Another useful operation of `Maybe` is `getOrElse` as an alternative to returning default values. Consider the example of setting the value of a form field, or a generic default in case there's no data to set:

```
const userName = findStudent('444-44-4444').map(R.prop('firstname'));

document.querySelector('#student-firstname').value =
    username.getOrElse('Enter first name');
```

If the fetch operation is successful, the student's username is displayed; otherwise, the `else` branch executes printing the default string.

> **Maybe in disguise**
>
> You may see `Maybe` appear in different forms such as the `Optional` or `Option` type, used in languages like Java 8 and Scala. Instead of `Just` and `Nothing`, these languages declare `Some` and `None`. Semantically, though, they do the same things.

Now let's revisit the pessimistic `null`-check anti-pattern shown earlier that rears its ugly head frequently in object-oriented software. Consider the `getCountry` function:

```
function getCountry(student) {
   let school = student.school();
   if(school !== null) {
      let addr = school.address();
      if(addr !== null) {
         return addr.country();
      }
   }
```

```
   return 'Country does not exist!';
}
```

What a drag. If the function returns `'Country does not exist!'`, which statement caused the failure? In this code, it's hard to discern which line is the problematic one. When you write code like this, you aren't paying attention to style and correctness; you're defensively patching function calls. Without monadic traits, you're basically stuck with `null` checks sprinkled all over the place to prevent `TypeError` exceptions. The `Maybe` structure encapsulates this behavior in a reusable manner. Consider this example:

```
const country = R.compose(getCountry, safeFindStudent);
```

Because `safeFindStudent` returns a wrapped student object, you can eliminate this defensive programming habit and safely propagate the invalid value. Here's the new `getCountry`:

```
const getCountry = (student) => student
      .map(R.prop('school'))
      .map(R.prop('address'))
      .map(R.prop('country'))
         .getOrElse('Country does not exist!');
```

**If any of the steps yields a Nothing result, all subsequent operations will be skipped.**

In the event that any of these properties returns `null`, this error is propagated through all the layers as a `Nothing`, so that all subsequent operations are gracefully skipped. The program is not only declarative and elegant, but also fault-tolerant.

### Function lifting

Look closely at this function:

```
const safeFindObject = R.curry((db, id) => {
   return Maybe.fromNullable(find(db, id));
});
```

Notice that its name is prefixed with `safe` and it uses a monad directly to wrap its return value. This is a good practice because you make it clear to the caller that the function is housing a potentially dangerous value. Does this mean you need to instrument every function in your program with monads? Not necessarily. A technique called *function lifting* can transform any ordinary function into a function that works on a container, making it "safe." It can be a handy utility so that you aren't obligated to change your existing implementations:

```
const lift = R.curry((f, value) => {
   return Maybe.fromNullable(value).map(f);
});
```

Instead of directly using the monad in the body of the function, you can keep it as is

```
const findObject = R.curry((db, id) => {
    return find(db, id);
});
```

and use `lift` to bring this function into the container:

```
const safeFindObject = R.compose(lift(console.log), findObject);
safeFindObject(DB('student'), '444-44-4444');
```

Lifting can work with any function on any monad!

Clearly, `Maybe` excels at centrally managing checks for invalid data, but it provides `Nothing` (pun intended) with regard to what went wrong. We need a more proactive solution—one that can let us know the cause of the failure. For this, the best tool to use is the `Either` monad.

#### RECOVERING FROM FAILURE WITH EITHER

`Either` is slightly different from `Maybe`. `Either` is a structure that represents a logical separation between two values `a` and `b` that would never occur at the same time. This type models two cases:

- `Left(a)`—Contains a possible error message or throwable exception object
- `Right(b)`—Contains a successful value

`Either` is typically implemented with a bias on the right operand, which means mapping a function over a container is always performed on the `Right(b)` subtype. It's analogous to the `Just` branch of `Maybe`.

A common use of `Either` is to hold the results of a computation that may fail to provide additional information as to what the failure is. In unrecoverable cases, the left can contain the proper exception object to throw. The following listing shows the implementation of the `Either` monad.

**Listing 5.6 `Either` monad with `Left` and `Right` subclasses**

```
class Either {
  constructor(value) {
     this._value = value;
  }

  get value() {
     return this._value;
  }

  static left(a) {
    return new Left(a);
  }
```

**Constructor function for either type. This can hold an exception or a successful value (right bias).**

```
  static right(a) {
    return new Right(a);
  }

  static fromNullable(val) {
     return val !== null && val !== undefined ? Either.right(val) :
     Either.left(val);
  }

  static of(a){
     return Either.right(a);
  }
}

class Left extends Either {

   map(_) {
    return this; // noop
   }

   get value() {
      throw new TypeError("Can't extract the
         value of a Left(a).");
   }

   getOrElse(other) {
      return other;
   }

   orElse(f) {
      return f(this._value);
   }

   chain(f) {
      return this;
   }

   getOrElseThrow(a) {
      throw new Error(a);
   }

   filter(f) {
      return this;
   }

   toString() {
     return `Either.Left(${this._value})`;
   }
 }

class Right extends Either {
    map(f) {
       return Either.of(f(this._value));
    }

    getOrElse(other) {
       return this._value;
    }
```

**Takes the Left case with an invalid value, or else the Right**

**Creates a new instance holding a value on the Right**

**Transforms the value on the Right structure by mapping a function to it; does nothing on the Left**

**Extracts the Right value of the structure if it exists; otherwise, produces a TypeError**

**Extracts the Right value; if it doesn't have one, returns the given default**

**Applies a given function to a Left value; does nothing on the Right**

**Applies a function to a Right and returns that value; does nothing on the Left. This is the first time you encounter chain (explained later).**

**Throws an exception with the value only on the Left structure; otherwise, ignores the exception and returns the valid value**

**If a value is present and meets the given predicate, returns a Right describing the value; otherwise returns an empty Left**

**Transforms the value on the Right structure by mapping a function to it; does nothing on the Left**

**Extracts the Right value; if it doesn't have one, returns the given default**

```
    orElse() {
       return this;
    }

    chain(f) {
       return f(this._value);
    }

    getOrElseThrow(_) {
       return this._value;
    }

    filter(f) {
        return Either.fromNullable(f(this._value) ? this._value : null);
    }

    toString() {
        return `Either.Right(${this._value})`;
    }
}
```

**Applies a given function to a Left value; does nothing on the Right**

**Applies a function to a Right and returns that value; does nothing on the Left. This is the first time you encounter chain (explained later).**

**Throws an exception with the value only on the Left structure; otherwise, ignores the exception and returns the valid value**

**If a value is present and meets the given predicate, returns a Right describing the value; otherwise returns an empty Left**

Notice in both the `Maybe` and `Either` types that some operations are empty (no-op). These are deliberate and are meant to act as placeholders that allow functions to safely skip execution when the specific monad deems appropriate.

Now, let's put `Either` to use. This monad offers another alternative for the `safeFindObject` function:

```
const safeFindObject = R.curry((db, id) => {
   const obj = find(db, id);
   if(obj) {
      return Either.of(obj);
   }
   return Either.left(`Object not found with ID: ${id}`);
});
```

**Could also use Either.fromNullable() to abstract the entire if-else statement. I did it this way for illustration purposes.**

**The Left structure can hold values as well.**

If the data access operation is successful, a student object is stored in the right side (biased to the right); otherwise, an error message is provided on the left, as shown in figure 5.9.

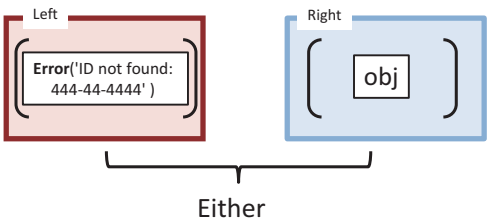


**Figure 5.9 An `Either` structure can store an object (on the right) or an `Error` (on the left) with proper stack trace information. This is useful to provide a single return value that can also contain an error message in case of failure.**

Let's pause for a second. You may be wondering, "Why not use the 2-tuple (or a `Pair`) type discussed in chapter 4 to capture the object and a message?" There's a subtle reason. Tuples represent what's known as a *product type*, which implies a logical AND relationship among its operands. In the case of error handling, it's more appropriate to use mutually exclusive types to model the case of a value either existing OR not; in the case of error handling, both could not exist simultaneously.

With `Either`, you can extract the result by calling `getOrElse` (providing a suitable default just in case):

```
const findStudent = safeFindObject(DB('student'));
findStudent('444-44-4444').getOrElse(new Student()); //->Right(Student)
```

Unlike the `Maybe.Nothing` structure, the `Either.Left` structure can contain values to which functions can be applied. If `findStudent` doesn't return an object, you can use the `orElse` function on the `Left` operand to log the error:

```
const errorLogger = _.partial(logger, 'console', 'basic', 'MyErrorLogger',
    'ERROR');
findStudent('444-44-4444').orElse(errorLogger);
```

This prints to the console:

```
MyErrorLogger [ERROR] Student not found with ID: 444-44-4444
```

The `Either` structure can also be used to guard your code against unpredictable functions (implemented by you or someone else) that may throw exceptions. This makes your functions more type-safe and side effect–free by eliminating the exception early on instead of propagating it. Consider an example using JavaScript's `decodeURIComponent` function, which can produce a URI error if it's invalid:

```
function decode(url) {
   try {
      const result = decodeURIComponent(url);
      return Either.of(result);
   }
   catch (uriError) {
      return Either.Left(uriError);
   }
}
```

**Throws a URIError.**

As shown in this code, it's also customary to populate `Either.Left` with an error object that contains stack trace information as well as an error message; this object can be thrown if necessary to signal an unrecoverable operation. Suppose you want to navigate to a given URL that needs to be decoded first. Here's the function invoked with invalid and valid input:

```
const parse = (url) => url.parseUrl();
decode('%').map(parse); //-> Left(Error('URI malformed'))
decode('http%3A%2F%2Fexample.com').map(parse);
//-> Right(true)
```

**This function was created in section 4.4.1.**

Functional programming leads to avoiding ever having to throw exceptions. Instead, you can use this monad for lazy exception throwing by storing the exception object into the left structure. Only when the left structure is unpacked does the exception take place:

```
...

catch (uriError) {
   return Either.Left(uriError);
}
```

Now you've learned how monads help emulate a `try-catch` mechanism that contains potentially hazardous function calls. Scala implements a similar notion using a type called `Try`—the functional alternative to `try-catch`. Although not fully a monad, `Try` represents a computation they may either result in an exception or return a fully computed value. It's semantically equivalent to `Either`, and it involves two cases classes for `Success` and `Failure`.

> **Functional programming projects worth exploring**
>
> Most of the topics explored in this and the previous chapter, such as partial application, tuples, composition, functors, and monads, as well as other topics presented later, are implemented as modules in a formal specification called Fantasy Land (https://github.com/fantasyland). Fantasy Land is a reference implementation of functional concepts that defines how to implement a functional algebra in JavaScript. We've been using libraries like Lodash and Ramda for their ease of use; nevertheless, Fantasy Land and a functional library called Folktale (http://folktalejs.org/) are worth exploring if you're eager to get deep into more-functional data types.

Monads can help you cope with uncertainty and possibilities for failure in real-world software. But how do you interact with the outside world?

### 5.3.3 Interacting with external resources using the IO monad

Haskell is believed to be the only programming language that relies heavily on monads for IO operations: file read/writes, writing to the screen, and so on. You can translate that to JavaScript with code that looks like this:

```
IO.of('An unsafe operation').map(alert);
```

Although this is a simple example, you can see intricacies of IO tucked into lazy monadic operations that are passed to the platform to execute (in this case, a simple alert message). But JavaScript unavoidably needs to be able to interact with the ever-changing, shared, stateful DOM. As a result, any operation performed on the DOM,

whether read or write, causes side effects and violates referential transparency. Let's begin with most basic IO operations:

```
const read = (document, selector) => {
   return document.querySelector(selector).innerHTML;
};

const write = (document, selector, val) => {
   document.querySelector(selector).innerHTML = val;
   return val;
};
```

**Subsequent calls to read may yield different results.**

**Doesn't return a value, and clearly causes mutations to happen (unsafe operation).**

When executed independently, the output of these standalone functions can never be guaranteed. Not only does order of execution matter, but, for instance, calling `read` multiple times can yield different responses if the DOM was modified between calls by another call to `write`. Remember, the main reason for isolating impure behavior from pure code, as you did in chapter 4 with `showStudent`, is to always guarantee a consistent result.

You can't avoid mutations or fix the problem with side effects, but you can at least work with IO operations as if they were immutable from the application point of view. This can be done by lifting IO operations into monadic chains and letting the monad drive the flow of data. To do so, you can use the `IO` monad.

**Listing 5.7 `IO` monad**

```
class IO {
   constructor(effect) {
      if (!_.isFunction(effect)) {
         throw 'IO Usage: function required';
      }
      this.effect = effect;
   }

   static of(a) {
      return new IO( () => a );
   }

   static from(fn) {
      return new IO(fn);
   }

   map(fn) {
       let self = this;
       return new IO(() => fn(self.effect()));
   }

   chain(fn) {
        return fn(this.effect());
   }
```

**The IO constructor is initialized with a read/write operation (like reading or writing to the DOM). This operation is also known as the effect function.**

**Unit functions to lift values and functions into the IO monad**

**Map functor**

```
    run() {
         return this.effect();
    }
}
```

**Kicks off the lazily initialized chain to perform the IO**

This monad works differently than the others, because it wraps an *effect* function instead of a value; remember, a function can be thought of as a *lazy value*, if you will, waiting to be computed. With this monad, you can chain together any DOM operations to be executed in a single "pseudo" referentially transparent operation and ensure that side effect–causing functions don't run out of order or between calls.

Before I show you this, let's refactor `read` and `write` as manually curried functions:

```
const read = (document, selector) => {
  return () => {
    return document.querySelector(selector).innerHTML;
  };
};

const write = (document, selector) => {
  return (val) => {
    document.querySelector(selector).innerHTML = val;
    return val;
  };
};
```

And in order to avoid passing the `document` object around, make life easier and partially apply it to these functions:

```
const readDom = _.partial(read, document);
const writeDom = _.partial(write, document);
```

With this change, both `readDom` and `writeDom` become chainable (and composable) functions awaiting execution. You do this in order to chain these IO operations together later. Consider a simple example that reads a student's name from an HTML element and changes it to start-case (capitalize the first letter of each word):

```
<div id="student-name">alonzo church</div>

const changeToStartCase =
    IO.from(readDom('#student-name'))
        .map(_.startCase)
        .map(writeDom('#student-name'));
```

**You can map any transformation operation here.**

Writing to the DOM, the last operation in the chain, isn't pure. So what do you expect the `changeToStartCase` output to be? The nice thing about using monads is that you preserve the requirements imposed by pure functions. Just like any other monad, the output from `map` is the monad itself, an instance of `IO`, which means at this stage nothing has been executed yet. What you have here is a declarative description of an IO operation. Finally, let's run this code:

```
changeToStartCase.run();
```

Inspecting the DOM, you'll see this:

```
<div id="student-name">Alonzo Church</div>
```

There you have it: IO operations in a referentially transparent-ish way! The most important benefit of the `IO` monad is that it clearly separates the pure and impure parts. As you can see in the definition of `changeToStartCase`, the transformation functions that map over the IO container are completely isolated from the logic of reading and writing to the DOM. You can transform the contents of the HTML element as needed. Also, because it all executes in one shot, you guarantee that nothing else will happen between the read and write operations, which can lead to unpredictable results.

Monads are nothing more than chainable expressions or chainable computations. This allows you to build sequences that apply additional processing at each step—like a conveyor belt in an assembly line. But chaining operations isn't the only modality where monads are used. Using monadic containers as return types creates consistent, type-safe return values for functions and preserves referential transparency. Recall from chapter 4 that this satisfies the requirement for composing function chains and compositions.

## 5.4 Monadic chains and compositions

As you can see, monads bring the world of side effects under control, so you can use them in composable structures. As you know from chapter 4, compositionality is the trick to reducing complexity in your code. But in chapter 4, you hadn't bothered to check for invalid data: if `findStudent` had returned `null`, the entire program would have failed, as shown in figure 5.10.

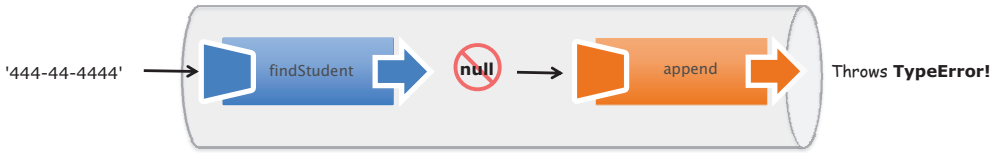


**Figure 5.10 Functions `findStudent` and `append` are being composed. Without the proper checks, if the former produces a `null` return value, the latter will fail with a `TypeError` exception.**

Fortunately, with little code, monads can also be made composable so that you can enjoy their fluent, expressive error-handling mechanism to create safe compositions. Wouldn't it be nice if functions arranged in a pipeline gracefully sidestepped `null` mines?

As you can see in figure 5.11, the first step is to make sure the first function to be executed wraps its result in a proper monad: both `Maybe` and `Either` work in this case.

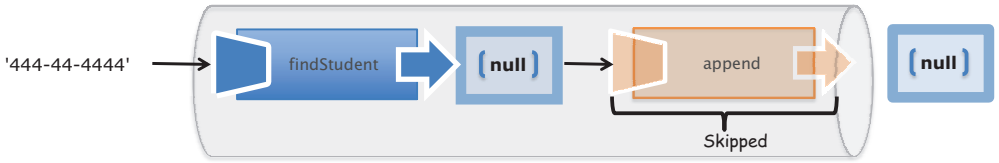


**Figure 5.11 Same two functions as in figure 5.10; but this time the `null` value travels in a monad (`Either` or `Maybe`), which causes the rest of the functions in the pipeline to gracefully fail.**

As you know, there are two variations for combining functions in functional programming: chain and compose. Recall that `showStudent` from the previous chapter had three parts:

1. Normalize user input
2. Find the student record
3. Add the student information to the HTML page

You're also adding input validation to the mix to make it even more complex. Hence, this program has two points of failure: a validation error and an unsuccessful student-fetch operation. You can refactor them to include the `Either` monad to supply appropriate error messages, as shown next.

**Listing 5.8 Refactoring functions to use `Either`**

```
// validLength :: Number, String -> Boolean
const validLength = (len, str) => str.length === len;

// checkLengthSsn :: String -> Either(String)
const checkLengthSsn = ssn => {
    return Either.of(ssn)
      .filter(R.partial(validLength, [9]));
};

// safeFindObject :: Store, string -> Either(Object)
const safeFindObject = R.curry((db, id) => {
    return Either.fromNullable(find(db, id));
});

// finStudent :: String -> Either(Student)
const findStudent = safeFindObject(DB('students'));

// csv :: Array => String
const csv = arr => arr.join(',');
```

**Instead of lifting these functions into an Either, you can use the monad directly and provide specific error messages depending on the error.**

**Refactored csv function returns a string from an array of values**

Because these functions are curried, you can partially evaluate them to create simpler ones, as you did before, as well as add some helper logging functions:

```
const debugLog = _.partial(logger, 'console', 'basic',
     'Monad Example', 'TRACE');

const errorLog = _.partial(logger, 'console', 'basic',
     'Monad Example', 'ERROR');

const trace = R.curry((msg, val)=> debugLog(msg + ':' + val));
```

And that's it! The monadic operations take care of the rest and ensure that the data travels through the function calls at no additional cost. Let's look at how you can use `Either` and `Maybe` to add automatic error handling to `showStudent`.

**Listing 5.9 `showStudent` using monads for automatic error handling**

```
const showStudent = (ssn) =>
   Maybe.fromNullable(ssn)
      .map   (cleanInput)
      .chain(checkLengthSsn)
      .chain(findStudent)
      .map   (R.props(['ssn', 'firstname', 'lastname']))
      .map   (csv)
      .map   (append('#student-info'));
```

**Methods map and chain can be used to transform the value in the monad. Map returns a monad; to avoid nesting and having to flatten the structure, weave map with chain to keep a single monad level flowing through the calls.**

**Extracts the selected properties from an object as an array**

Listing 5.9 shows the use of the `chain` method. This is nothing more than a shortcut to avoid having to use `join` after `map` to flatten the layers resulting from combining monad-returning functions. Like `map`, `chain` applies a function to the data without wrapping the result back into the monad type.

Also, notice how both monads interleave seamlessly. This is because both `Either` and `Maybe` implement the same monadic interface. Now, calling

```
showStudent('444-44-4444').orElse(errorLog);
```

generates two results: if the student object is successfully found, it appends the student information to the HTML as expected and returns:

```
Monad Example [INFO] Either.Right('444-44-4444, Alonzo,Church')
```

Otherwise, it skips the entire operation gracefully and uses the `orElse` clause:

```
Monad Example [ERROR] Student not found with ID: 444444444
```

Chaining isn't the only pattern; you can easily introduce error-handling logic with compose. To do this, you perform the simple object-oriented-to-functional transform you've seen before to convert monad methods into functions that polymorphically

work across any monad type (following from the Liskov Substitution Principle). In particular, you can create generalized `map` and `chain` functions, shown in the following listing.

**Listing 5.10 General `map` and `chain` functions that work on any container**

```
// map :: (ObjectA -> ObjectB), Monad -> Monad[ObjectB]
const map = R.curry((f, container) => {
    return container.map(f);
});

// chain :: (ObjectA -> ObjectB), Monad -> ObjectB
const chain = R.curry((f, container) => {
    return container.chain(f);
});
```

You can use these functions to inject monads into a compose expression. The code in listing 5.11 produces the same results as listing 5.9. Because monads control how data flows from one expression to the next, this style of coding is also known as *programmable commas*, which is also point-free. In this case, a comma is used to delimit one expression from another in the same way a semicolon traditionally delineates one statement from the next in JavaScript. Also, using lots of `trace` statements lets you see the data flowing through the operations (logging statements are useful for debugging, as well).

**Listing 5.11 Monads as programmable commas**

```
const showStudent = R.compose(
     R.tap(trace('Student added to HTML page'))
     map(append('#student-info')),
     R.tap(trace('Student info converted to CSV')),
     map(csv),
     map(R.props(['ssn', 'firstname', 'lastname'])),
     R.tap(trace('Record fetched successfully!')),
     chain(findStudent),
     R.tap(trace('Input was valid')),
     chain(checkLengthSsn),
     lift(cleanInput));
```

Running the code prints the following log messages on the console:

```
Monad Example [TRACE] Input was valid:Either.Right(444444444)

Monad Example [TRACE] Record fetched successfully!: Either.Right(Person
[firstname: Alonzo| lastname: Church])

Monad Example [TRACE] Student converted to CSV: Either.Right(444-44-4444,
Alonzo, Church)

Monad Example [TRACE] Student added to HTML page: Either.Right(1)
```

### Tracing through programs

Listing 5.11 demonstrates how easy it is to trace through functional code. Without having to drill into the body of those functions, you can demarcate an entire program with tracing statements that execute before and after function calls, which is incredibly useful for troubleshooting and debugging. If this program were written in an object-oriented style, you couldn't possibly do this without having to modify the actual functions or perhaps instrument them using aspect-oriented programming, which isn't a trivial endeavor. Functional programming gives you this for free!

Finally, let's diagram this entire flow to clearly see what's going on; see figure 5.12. Figure 5.13 shows the behavior of this same program in the event that `findStudent` is unsuccessful.

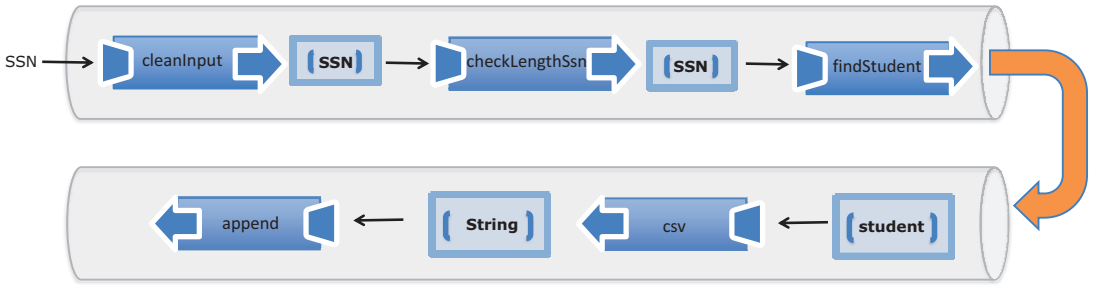


**Figure 5.12 Step-by-step flow of the `showStudent` function in the case where `findStudent` successfully finds a student object by the provided SSN**

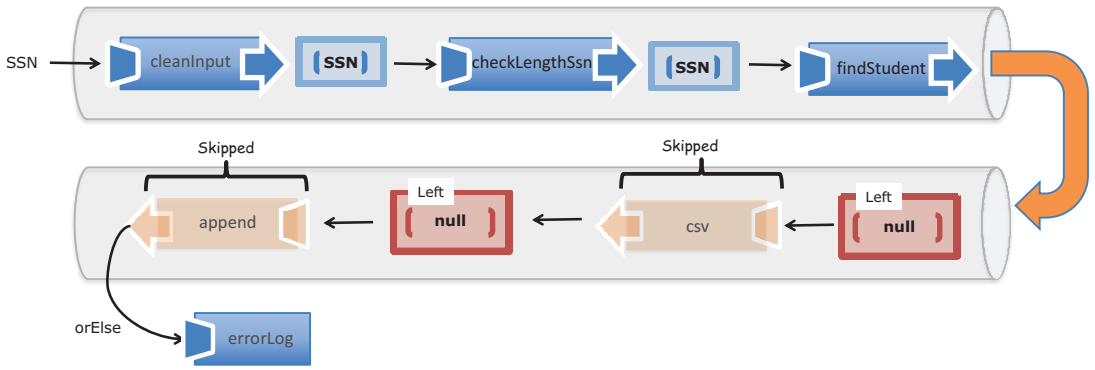


**Figure 5.13 The case of an unsuccessful `findStudent` as it affects the rest of the composition. Regardless of the failure of any of the components in the pipeline, the program remains fault-tolerant and gracefully skips any procedures that depended on the data.**

You may be wondering if you're finally done with `showStudent`. Not quite. From the discussion of the `IO` monad, now you know you can improve the code that deals with DOM reads and writes:

```
map(append('#student-info')),
```

Because `append` has automatic currying, it'll work well with `IO`. All you need to do at this point is lift the value from `csv`, extract its content by mapping the `R.identity` function into `IO` using `IO.of`, and then proceed with chaining both operations:

```
const liftIO = function (val) {
   return IO.of(val);
};
```

This produces the following program.

**Listing 5.12 Complete `showStudent` program**

```
const showStudent = R.compose(
   map(append('#student-info')),
   liftIO,
   map(csv),
   map(R.props(['ssn', 'firstname', 'lastname'])),
   chain(findStudent),
   chain(checkLengthSsn),
   lift(cleanInput));
```

Incorporating the `IO` monad allows you to achieve something truly amazing. You see, running `showStudent(ssn)` now runs through all the logic of validating and fetching the student record, as it should. Once this completes, the program waits on you to write this data to the screen. Because you've lifted the data into an IO monad, you need to call its `run` function for the data that's lazily contained within it (in its closure) to be flushed out to the screen:

```
showStudent(studentId).run(); //-> 444-44-4444, Alonzo, Church
```

A common pattern that occurs with `IO` is to tuck the impure operation toward the end of the composition. This lets you build programs one step at a time, perform all the necessary business logic, and finally deliver the data on a silver platter for the `IO` monad to finish the job, declaratively and side effect–free.

Just to show how functional programming makes code easier to reason about, for the sake of comparison (apologies for reviving some ugly code), let's bring back the equivalent nonfunctional version of `showStudent`:

```
function showStudent(ssn) {
    if(ssn != null) {
        ssn = ssn.replace(/^\s*|\-|\s*$/g, '');
        if(ssn.length !== 9) {
            throw new Error('Invalid Input');
        }
```

```
        let student = db.get(ssn);
        if (student) {
            document.querySelector(`#${elementId}`).innerHTML =
              `${student.ssn},
               ${student.firstname},
               ${student.lastname}`;
        }
        else {
            throw new Error('Student not found!');
        }
    }
    else {
        throw new Error('Invalid SSN!');
    }
}
```

Due to side effects, lack of modularity, and imperative error handling, this program is difficult to use and test; we'll examine this more closely in the next chapter. Whereas composition controls program flow, monads control data flow. Both are possibly the most important concepts in the functional programming ecosystem.

This chapter completes part 2 of the book. Your developer toolbox is equipped with all the functional concepts you need to take on real-world solutions.

## 5.5 Summary

- Exception-throwing mechanisms in object-oriented code result in impure functions that impose a great deal of responsibility on the caller to provide adequate `try-catch` logic.
- The pattern of value containerization is used to create side effect–free code by wrapping possible mutations under a single referentially transparent process.
- Use functors to map functions to containers in order to access and modify objects in a side effect–free and immutable manner.
- Monads are a functional programming design pattern used to reduce an application's complexity by orchestrating a secure flow of data through functions.
- Resilient and robust function compositions interleave monadic types such as `Maybe`, `Either`, and `IO`.

JAVASCRIPT

# Functional Programming in JavaScript

Luis Atencio


In complex web applications, the low-level details of your JavaScript code can obscure the workings of the system as a whole. As a coding style, functional programming (FP) promotes loosely coupled relationships among the components of your application, making the big picture easier to design, communicate, and maintain.

**Functional Programming in JavaScript** teaches you techniques to improve your web applications—their extensibility, modularity, reusability, and testability, as well as their performance. This easy-to-read book uses concrete examples and clear explanations to show you how to use functional programming in real life. If you're new to functional programming, you'll appreciate this guide's many insightful comparisons to imperative or object-oriented programming that help you understand functional design. By the end, you'll think about application design in a fresh new way, and you may even grow to appreciate monads!

## What's Inside

- High-value FP techniques for real-world uses
- Using FP where it makes the most sense
- Separating the logic of your system from implementation details
- FP-style error handling, testing, and debugging
- All code samples use JavaScript ES6 (ES 2015)

Written for developers with a solid grasp of JavaScript fundamentals and web application design.

**Luis Atencio** is a software engineer and architect building enterprise applications in Java, PHP, and JavaScript.

"This book transformed the way that I think about and write JavaScript."
—Andrew Meredith
Intrinsitech Corporation

"Easy to navigate, with real-life examples."
—Amy Teng, Dell

"Now, this is the way to write JavaScript!"
—William E. Wheeler, West Corporation

"After reading this book, I revisited how I approached coding and was able to retrain my mind using better methods and techniques."
—Tanner Slayton Sr.
Microsoft Corporation

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/functional-programming-in-javascript

ISBN-13: 978-1-61729-282-8
ISBN-10: 1-61729-282-6
54499
9 781617 292828

MANNING US $ 44.99 | Can $ 51.99 (eBook included)