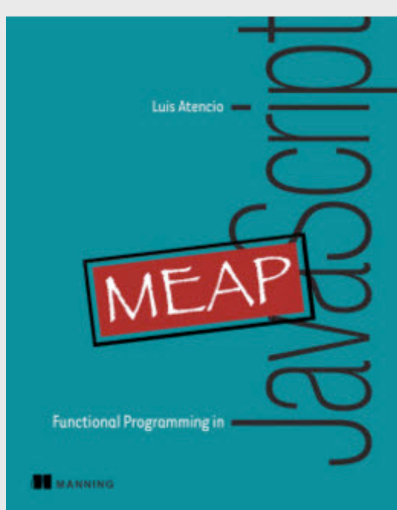

Functional error handling using monads

By Luis Atencio

In this article, excerpted from Functional Programming in JavaScript, I introduce you to monads and explain when and how to use them.

Monads solve the problems of traditional error handling when applied to functional programs. But before diving into this topic, let's first understand a limitation in the use of functors. You can use functors to safely apply functions onto values in an immutable and safe manner. However, 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, we can identify two functions: `findStudent` and `getAddress`, both using functor objects to create a safe context around their returned values

```
var findStudent = R.curry(function(db, ssn) {
   return wrap(find(db, ssn)); //#A
});

var getAddress = function(student) {
   return wrap(student.fmap(R.prop('address'))); //#B
}
```

#A - Wrapped the fetched object to safeguard against the possibility of not finding an object
#B - Mapping Ramda's R.prop() function over the object to extract its address, and then wrapping the result

To run this program I'll compose both functions together

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

While I was able to avoid all error handling code, the result is not what I originally expected. 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, I would have to apply `R.identity` twice

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

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

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` over any wrapped value

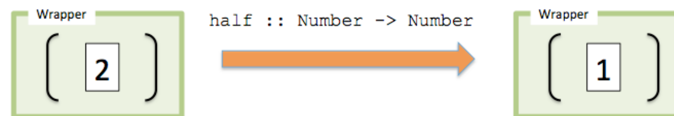

Figure 1 Functors apply a function onto a wrapped value. In this case, the wrapped value 2, is halved, returning a wrapped value of 1

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

But, now suppose I wanted 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. So, what can I do if I encounter an odd input value? I 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 ignore it otherwise.

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

```
var Empty = function (_) {
   ;  //#A
};

// map :: (A -> B) -> A -> B
Empty.prototype.map = function() { return this; };  //#B

// empty :: _ -> Empty
var empty = () => new Empty();
```

#A - noop; Empty does not store a value, it represents the concept of "empty" or "nothing"
#B - Similarly, mapping a function onto an empty just skips the operation

With this new requirement, I can implement `half` in the following way

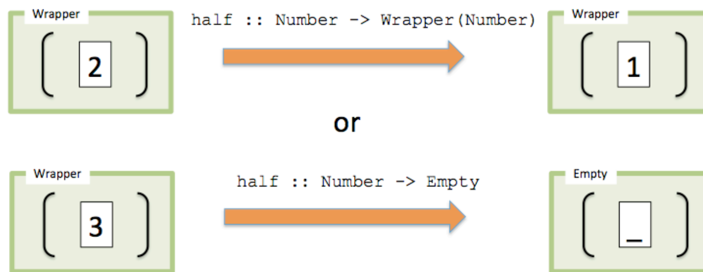

Figure 2 Function `half` can return either a wrapped value or an empty container depending on the nature of the input

```
var isEven = (n) => Number.isFinite(n) && (n % 2 == 0); //#A

var half = (val) => isEven(val) ? wrap(val / 2) : empty(); //#B


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

#A - Helper function used to distinguish between odd and even numbers
#B - Function half only works on even numbers, returning an empty container otherwise

For source code, sample chapters, the Online Author Forum, and other resources, go to
https://www.manning.com/books/functional-programming-in-javascript

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

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

#A - The implicit container knows how to map functions even when input is invalid

Monads can be targeted at a variety of problems; but the ones we will study in this article 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 very dependent on the type system of a language. In fact, many people advocate you can only understand and them if you have explicit types, like in Haskell. But I'll demonstrate that having a type-less language like JavaScript actually makes them easy to read and frees you from having to deal with all of the intricacies of a static type system.

There are two important concepts you need to understand:

- **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 with the `Wrapper` object. However, every monad is different and, depending on its purpose, can define different semantics driving its behavior (i.e., 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:

- A *type constructor*: used for creating monadic types (similar to the `Wrapper` constructor)
- A *unit* function: used for inserting a value of a certain type into a monadic structure. When implemented in the monad, though, this function is called `of`.
- A *bind* function: used for chaining operations together (this is actually a functor's

fmap, also known as flatMap).

- A *join* operation: used to flatten layers of monadic structures into one, especially important when composing composing multiple monad-returning functions.

Applying this new interface to the Wrapper type, I can refactor it in the following way:

Listing 1 The Wrapper monad

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

    static of(a) { //#B
       return new Wrapper(a);
    }

    map(f) {//#C
      return Wrapper.of(f(this.value));
    }

    join() {//#D
      if(!(this.value instanceof Wrapper)) {
         return this;
      }
      return this.value.join();
    }

    toString() {  //#E
      return `Wrapper (${this.value})`;
    }
 }
```

#A - Type constructor
#B - The unit function
#C - The bind function (the functor)
#D - Flatten nested layers

Wrapper uses the functor map to lift data into the container so that I 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* since it does nothing more than map the function and close it back. Later, you'll see other monads add their own special touches to `map`. Moreover, the `join` function is used to to flatten nested structures—like peeling back an onion. This can be used to eliminate the issues found with functors earlier

Listing 2 Flattening a monadic structure

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

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

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

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

Because the composition in listing 2 returns a set of nested wrappers, the `join` operation was used to flatten out the structure into a single layer, like in this trivial example:

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

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

We can visualize the join operation with figure 3:

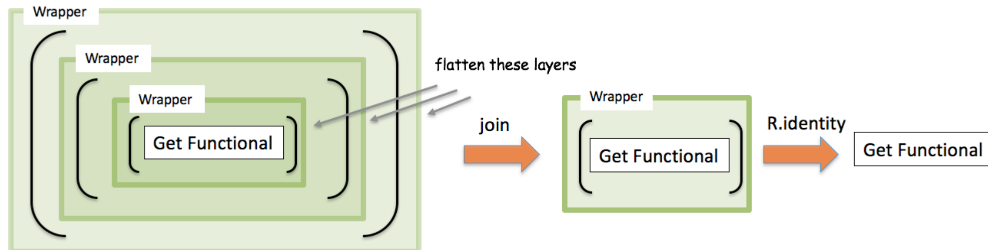

Figure 3 Using the `join` operation to recursively flatten a nested monad structure like peeling back an onion

In relation with arrays (which are also just containers that can be mapped on), this is analogous to the `R.flatten` operation:

For source code, sample chapters, the Online Author Forum, and other resources, go to
https://www.manning.com/books/functional-programming-in-javascript

```
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 its 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 is when 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 we will look at some full-fledge monads:

1. Maybe and Either
2. IO

Error handling with Maybe and Either monads

Aside from 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 in order to:

- 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 will begin with the `Maybe` monad.

CONSOLIDATING NULL-CHECKS WITH MAYBE

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

- `Just(value)`: represents a container that wraps a defined value.
- `Nothing()`: represents a container that has no value, or a failure that needs no additional information. In the case of a `Nothing`, you are still able to apply functions over its (in this case non-existent) value.

These subtypes implement all of the monadic properties, as well as some additional behavior unique to its purpose. Here's an implementation of `Maybe` in listing 3:

Listing 3 The Maybe monad with subclasses Just and Nothing

```
class Maybe {  //#A
```

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

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

  static fromNullable(a) {
    return a !== null ? just(a) : nothing(); //#B
  }

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

  get isNothing() {
     return false;
  }

  get isJust() {
     return false;
  }
}

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

   get value() {
      return this._value;
   }

   map(f) {
      return of(f(this.value)); //#D
   }

   getOrElse() {
     return this.value; //#E
   }

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

   get isJust() {
     return true;
   }

   toString () { //#F
     return `Maybe.Just(${this.value})`;
   }
```

```
}

class Nothing extends Maybe { //#G

   map(f) {
    return this; // noop (mapping over nothing)
   }

   get value()  {
     throw new TypeError('Can't extract the value
        of a Nothing.'); //#H
   }

   getOrElse(other) {
     return other; //#I
   }


   filter() {
     return this.value; //#J
   }

   get isNothing() {
      return true;
   }

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

#A - Container type (parent class)
#B - Builds a Maybe from a nullable type (constructor function). If the value lifted in the monad is null, it instantiates a Nothing; otherwise, it stores the value in a Just subtype to handle the presence of a value
#C - Subtype Just to handle the presence of a value
#D - Mapping a function over a Just, applies the function onto a value and stores it back into the container
#E - Extract the value from the structure, or a provided default monad unity operation
#F - Returns a textual representation of this structure
#G - Subtype Nothing to handle the protect against the absence of a value
#H - Attempting to extract a value from a Nothing type generates an exception indicating a bad use of the monad (I will discuss this shortly); otherwise, the value is returned
#I - Ignore the value and return the other
#J- If a value is present, and the value matches the given predicate, return a Just describing the value, otherwise return a Nothing

`Maybe` explicitly abstracts working with "nullable" values (`null` and `undefined`) so that 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 their 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 4.

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

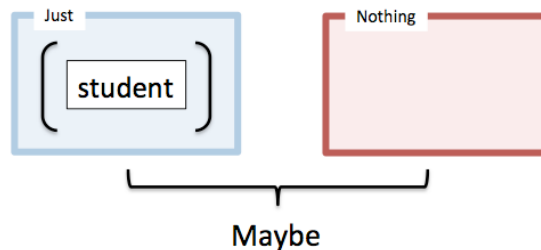

Figure 4 A Maybe structure has two subtypes: `Just` and `Nothing`. Calling `findStudent` returns its value wrapped in `Just`, or the absence of one in `Nothing`

This monad is frequently used with calls that contain uncertainty: querying a database, looking up values in a collection, requesting data from the server, etc. I'll continue with the example started in listing 2 of extracting the `address` property of a student object that's fetched from a local store. Because a record might or might not exist, I will wrap the result of the fetch in a `Maybe` and add the "safe" prefix to these operations

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

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

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

Another benefit to wrapping results with monads is that it embellishes your function signature making it self-documented and honest about the uncertainty of its return value. `Maybe.fromNullable` is very useful as it handles the null checking on our behalf. Calling `safeFindStudent` will either produce a `Just(Address(...))` in case it encounters a valid value or a `Nothing`, otherwise. So that mapping `R.prop` over the monad behaves as

expected. In addition, it also does a good job at detecting programmatic errors or misuses of an API call because you can use it to enforce pre-conditions indicating whether parameters are permitted to be invalid. If an invalid value is passed into `Maybe.fromNullable`, it will produce a `Nothing` type, such that calling `get()` to open up the container will throw an exception.

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

Monads expect you to stick to mapping functions over it instead of directly extracting its contents directly. Another very useful operation of `Maybe` is `getOrElse` as an alternative to returning default values. Consider this example displaying a student form first name field

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

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

If the fetch operation had been successful, the student's user name would have been 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, present in languages like Java 8 or Scala. In these languages, instead of declaring `Just` and `Nothing`, these are called `Some` and `None`. Semantically, however, both do the exact 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 function `getCountry` function:

```
function getCountry(student) {
   var school = student.school();
   if(school !== null) {
      var 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 are not really paying attention to style and correctness; you are simply defensively patching your 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:

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

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

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

#A - 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; your program is not only declarative and elegant, but also fault-tolerant.

Function lifting

If you look closer at this function:

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

You'll notice that I prefixed its name with `safe` and used a monad directly to wrap its return value. This is a good practice since I am making it clear to the caller that the function is housing a potentially dangerous value. Now, does this mean you need to

instrument every function in your program with monads? Not necessarily. There is a technique called *function lifting*. Lifting can transform any ordinary function into a function that works on a container, making it "safe." It can be a very handy utility so that you are not obligated to change your existing implementations.

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

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

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

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

```
   var safeFindObject = R.compose(lift, 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 really provides `Nothing` (pun intended) with regards to what went wrong. We need a more proactive solution, one that can let us know what the cause of the failure is. For this, the best tool to use is the `Either` monad.

RECOVERING FROM FAILURE WITH EITHER

`Either` is slightly different from a `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 different 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 that mapping a function over a container will always be performed on the `Right(b)` subtype—it is analogous to the `Just` branch of a `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, perhaps the left can contain the proper exception object to throw.

Listing 4 shows the implementation of the `Either` monad

Listing 4 The Either monad with Left and Right subclasses

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

    get value() {
       return this._value;
    }

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

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

    static fromNullable(val) { //#B
       return val !== null ? right(val): left(val);
    }


    static of(a){ //#C
       return right(a);
    }
}

class Left extends Either {

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

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

   getOrElse(other) {
      return other; //#F
   }

   orElse(f) {
      return f(this.value); //#G
   }

   chain(f) { //#H
      return this; // noop
   }

   getOrElseThrow(a) { //#I
      throw new Error(a);
   }
```

```
   filter(f) { //#J
      return this; // noop
   }

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

class Right extends Either {

    map(f) { //#D
       return Either.of(f(this.value));
    }

    getOrElse(other) {
      return this.value; //#F
    }

    orElse() { //#G
       return this; //noop
    }

    chain(f) { //#H
      return f(this.value);
    }

    getOrElseThrow(_) { //#I
      return this.value;
    }

    filter(f) { //#J
      return Either.fromNullable(f(this.value) ? this.value : null);
    }

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

#A - Constructor function for either type. This can hold an exception or a successful value (right bias)
#B - Takes the Left case with an invalid value; otherwise, the Right
#C - Creates a new instance holding a value on the Right
#D - Transforms the value on the Right structure by mapping a function onto it; does nothing on the Left
#E - Extracts the Right value of the structure if it exists; otherwise, produces a TypeError
#F - Extracts the Right value; if it doesn't have one it returns the given default
#G - Applies a given function onto a Left value; does nothing on the Right
#H - Applies a function onto a Right and returns that value; does nothing on the Left. This is the first time you encounter chain, and I'll explain its purpose later
 # I - Throws an exception with the value only on the Left structure; otherwise, the exception is ignored and the valid value returned

#J - If a value is present, and the value meets the given predicate, return a Right describing the value, otherwise return an empty Left

Notice in both `Maybe` and `Either` types that some operations are empty, a 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 our `safeFindObject` function:

```
var safeFindObject = R.curry(function (db, id) {
   var obj = find(db, id);
   if(obj) {
      return Either.of(obj);  //#A
   }
   return Either.Left(`Object not found with ID: ${id}`); //#B
});
```

#A - I could have also used Either.fromNullable() to abstract the entire if-else statement, but I did it this way for illustration purposes.
#B - The Left structure can hold values as well

If the data access operation is successful, a student object will be stored in the right side (biased to the right); otherwise, an error message is provided on the left, as shown in figure 5.

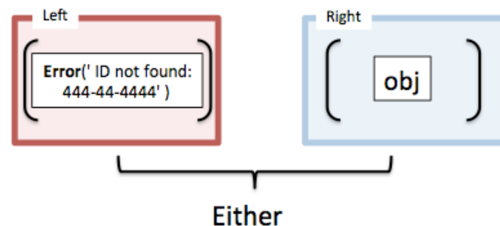

Figure 5 An `Either` structure can store an object (on the right) or an `Error` (on the left) with proper stack trace information. This is useful to provide a single return value that can also contain an error message in case of failure

Let me pause for a second here. You might be wondering, why not use the `2-Tuple` (or a `Pair`) type 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 amongst 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`, I can extract the result by calling `getOrElse` (providing a suitable default just in case).

```
var 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, I can use the `orElse` function on the `Left` operand to log the error:

```
var errorLogger = _.partial(logger, 'console', 'basic', 'MyErrorLogger',
'ERROR');

findStudent('444-44-4444').orElse(errorLogger);
```

Which will print 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 any unpredictable functions (implemented by you or someone else) that might 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 in case it's invalid:

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

As shown in the code above, 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 need be to signal an unrecoverable operation. Suppose we wanted to navigate to a given URL that needs to be decoded first. Here's the function invoked with invalid and valid input:

```
var parse = (url) => url.parseUrl(); //#A

decode('%').map(parse); //-> Left(Error('URI malformed'))

decode('http%3A%2F%2Fexample.com').map(parse);
//-> Right(true)
```

#A - This function was created in section 4.4.2

Functional programming tends to avoid ever having to throw exceptions. Instead, I could use this monad for lazy exception throwing by storing the exception object itself into the left structure. Only when the left structured is unpacked will 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`.

Indeed, monads can help us code with uncertainty and possibilities for failure in real world software, but how do we interact with the outside world?

Interact with external resources using the IO monad

It is believed that Haskell is the only programming language that relies heavily on monads for IO operations: such as file read/writes, writing to the screen, etc. We can translate that to JavaScript with code that looks like this

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

While this is a very simple example, you can see that the intricacies of IO are tucked into lazy monadic operations that are passed on to the platform to execute, in this case a simple alert message. However, JavaScript unavoidably needs to be able to interact with the ever-changing, shared, stateful DOM. As a result, any operation performed on it, whether read or write, will cause side effects and violate referential transparency. Let's begin with most basic IO operations

```
var read = function(document, id) {
   return document.querySelector(`\#${id}`).innerHTML; //#A
}

var write = function(document, id, val) {
   document.querySelector(`\#${id}`).innerHTML = value; //#B
};
```

#A - Subsequent calls to read might yield different results
#B - 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 in between calls by another call to `write`. Remember, the main reason for isolating the impure behavior from our pure code, as I did in chapter 4 with `showStudent`, is to always guarantee a consistent result.

While we can't avoid mutations or fix the problem with side effects, we 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. For this, we can use the IO monad

Listing 5 The IO monad

```
class IO {
  constructor(effect) { //#A
    if (!_.isFunction(effect)) {
        throw 'IO Usage: function required';
    }
    this.effect = effect;
  }
  static of(a) {  //#B
     return new IO( () => a );
```

```
    }
    static from(fn) { //#B
       return new IO(fn);
    }
    map(fn) { //#C
      var self = this;
      return new IO(function () {
          return fn(self.effect());
      });
    }
    chain(fn) {
       return fn(this.effect());
    }
    run() {
       return this.effect(); //#D
    }
 }
```

#A - The IO constructor is initialized with some read/write operation (like reading or writing to the DOM). This operations is also known as the effect function
#B - Unit functions to lift values and functions into the IO monad
#C - Map functor
#D - Kicks-off the lazily initialized chain to perform the IO

This monad works a bit 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 I 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 in between calls.

Before I show you this, I am going to refactor `read` and `write` as manually curried functions

```
var read = function (document, id) {
    return function () {
       return document.querySelector(`\#${id}`).innerHTML;
    };
```

```
};

var write = function(document, id) {
   return function(val) {
     return document.querySelector(`\#${id}`).innerHTML = val;
   };
};
```

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

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

With this change, both `readDom` and `writeDom` become chainable (and composable) functions awaiting execution. I do this in order to chain these IO operations together later. Consider a simple example that reads a student's name form a HTML element and changes it to start-case

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

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

#A - I can map any transformation operation here

Writing to the DOM, the last operation in the chain, is not pure. So, what do you expect the `changeToStartCase` output to be? The nice thing about using monads is that we still preserve the requirements imposed by pure functions. Indeed, 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 we have here is a declarative description of an IO operation. Finally, let's run this code

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

Inspecting the DOM you'll see:

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

There you have it, IO operations in a referentially transparent-ish way! Moreover, the most important benefit of the IO monad is that it clearly separates the pure from the 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. I could have transformed the contents of the HTML element as needed. Also, because it all executes in on shot, you guarantee that nothing else will happen in between the read and write operations, which can lead to unpredictable results.

Monads are really 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 is not the only modality where monads are used. Using monadic containers as return types creates consistent, type-safe return values for your functions and preserves referential transparency.

For source code, sample chapters, the Online Author Forum, and other resources, go to
https://www.manning.com/books/functional-programming-in-javascript