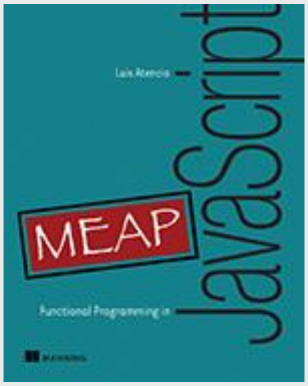


What is Functional Programming?

By By Luis Atencio

Functional programming is a software development style with emphasis on the use functions. It requires you to think a bit differently about how to approach tasks you are facing. In this article, based on the book Functional Programming in JavaScript, I will introduce you to the topic of functional programming.

The rapid pace of web platforms, the evolution of browsers, and, most importantly, the demand of your end users has had a profound effect in the way we design web applications today. Users demand web applications feel more like a native desktop or mobile app so that they can interact with rich and responsive widgets, which forces JavaScript developers to think more broadly about the solution domain push for adequate programming paradigms.

With the advent of Single Page Architecture (SPA) applications, JavaScript developers began jumping into object-oriented design. Then, we realized embedding business logic into the client side is synonymous with very complex JavaScript applications, which can get messy and unwieldy very quickly if not done with proper design and techniques. As a result, you easily accumulate lots of entropy in your code, which in turn makes your application harder write, test, and maintain. Enter Functional Programming.

Functional programming is a software methodology that emphasizes the evaluation of functions to create code that is clean, modular, testable, and succinct. The goal is to *abstract operations* on the data with functions in order to *avoid side effects* and *reduce mutation* of state in your application. However, thinking functionally is radically different from thinking in object-oriented terms. So, how do you become functional? How do you begin to think functionally? Functional programming is actually very intuitive once you've grasped its essence. Unlearning old habits is actually the hardest part as it can be a huge paradigm shift for most people that come from an object-oriented background. Let's convert a simple imperative "Hello World" program written in JavaScript, to functional:

Imperative

```
document.getElementById('msg').innerHTML = '<h1>Hello World<h1>';
```

Functional

```
compose(addToDom, h1, echo('Hello World'));
```

These smaller functions combine to implement a program that is easier to reason about as a whole. Why does the functional solution look this way? I like to think of it as basically parameterizing your code so that you can easily change it in a non-invasive manner. This is due to functional programming's inherent declarative mode of development. Imperative programming tells the computer, in great detail, *how* to perform a certain task. Declarative programming, on the other hand, separates program description from evaluation. It focuses on the use of *expressions* to describe *what* the logic of a program is without necessarily specifying its control flow or state changes.

This programming style is possible in JavaScript because of its support for first-class, high-order functions, which is a simple way of defining a function that accepts other functions as parameters or whose return value is a function.

While adopting a declarative style can be beneficial, sometimes you might find that functional code can be a bit harder write; this is a natural first impression before unlearning old methods and habits, as well as becoming comfortable with a *higher level of abstraction* or expressiveness. So why do it? The remarkable quality of declarative programs is that you wind up writing fewer lines of code and, simultaneously, get more work done. Consider this example: Write a short program to find the longest name in a collection of names. Your first approach is to write something like:

```
function longest(arr) {
    var longest = '';
    for (var i = 0; i < arr.length; i++) {
        if (arr[i].length > longest.length) {
            longest = arr[i];
        }
    }
    return longest;
}
```

This imperative approach manually iterates over the array from start to finish and keeps track of the longest state name as I iterate. Functional programming takes a different approach by raising the level of abstraction in your code and yielding most of the work to the most robust and optimized artifact in your application, your language runtime, without having to loop at all using *recursion*. Let's use it to refactor the same program:

```
function longest(str, arr) {
    if(isEmpty(arr)) {
       return str;
    else {
       var currentStr = head(arr).length >= str.length ? head(arr): str;
       return longest(currentStr, tail(arr));
    }
}
```

Why go through so much effort to eradicate loops from our code? Loops imply code that is constantly changing or mutating in response to new iterations. Functional programs aim for *statelessness* and *immutability* as much as possible so that after a section of code runs no existing variables are changed. To achieve this, functional programming makes use of functions that avoid *side effects* and changes of state—also known as *pure functions*.

PURE FUNCTIONS

Functional programming is based on the premise that you will build immutable programs solely based on pure functions*.* A pure function has the following qualities:

- Depends only on the input provided and not on any hidden or external state that may change as a function's evaluation proceeds or between function calls.
- Does not inflict changes beyond its scope, like modifying a global object or a parameter reference.

Pure functions can be very hard to use in a world full of dynamic behavior and mutation. But, practical functional programming doesn't restrict absolutely all state changes; it just provides a framework to help you manage and reduce, while allowing you separate the pure from the impure. It's more critical to ensure *no externally visible* side effects occur—immutability from the application's point of view.

Functions with side effects are, generally, bad design because they rely on data outside of the local scope of the function call, making it difficult to determine exactly what a function is supposed to do when they are run. Pure functions, on the other hand, have clear contracts as part of their signatures that describes clearly all of the function's formal parameters, making them simpler to understand.

Consider the example of fetching for processing a payment for student given their SSN. Let's model this with three side effect free functions.

```
var fetchStudent = curry(function (db, studentId) {

     return db.fetchStudentById(studentId);

});

var sendPayment = curry(function (payment, student) {
     return payment.submit(student)
});

sendNotification = curry(function (eventQueue, payment) {
     return eventQueue.fireEvent('Payment sent: '
             + payment.getAmount());
});
```

Composing all three functions makes up our `processPayment` function:

```
var processPayment = compose(
     sendNotification(eventQueue),
     sendPayment(paymentService),
     fetchStudent(Store('Students')));

processPayment('jims');
```

Removing all practical side effects made our programs less brittle to changing external conditions. Making a function's result consistent and predicable is a trait of pure functions called *referential transparency*.

REFERENTIAL TRANSPARENCY AND SUBSTITUTABILITY

Referential transparency is a more formal way of defining a pure function. Purity in this sense refers to the existence of a pure mapping between a function's arguments and its return value. Hence, if a function consistently yields the same result on the same input, it is said to be referentially transparent. Here's a quick example:

```
var incrementCounter = (counter) => counter + 1;
```

This function always returns the same output when provided the same input. If this were not the case, it must mean that that the function's return value is being influenced by some external factor. This quality of referential transparency is inherited from mathematical functions, which state that a function is a pure relation between inputs and outputs with the property that each input yields exactly one output.

Let’s tackle a more complex example: adding a new grade to a list and compute its average. I will create two functions `append` and `average`:

```
var append = function(grades, newGrade) {
    var newGrades = grades.slice(0);
    newGrades.push(newGrade);
    return newGrades;
}

var calcAverage = (grades) =>
   grades.reduce((total, current) => total + current) / grades.length;
```

Putting it all together:

```
var computeAverageGrade = run(Math.round, calcAverage, append);
```

This function yields a consistent result based only on the provided input and does not cause any side effects to the `grades` list after returning. Referential transparency is a very important quality of pure functions as it allows functional languages to optimize function calls, as you’ll see later in this chapter.

Furthermore, a corollary of referential transparency is the principle of *substitutability*. Because the result of a referentially transparent functions consistently depends only on the values of the arguments provided to it, we say that a function’s return value can be directly replaced into an expression without changing its meaning. Math operators behave this way: adding `2 + 3` can be substituted with `5` into any equation and the result of the equation will forever remain the same. The following statements are all equivalent:

```
(2 + 3) * 8 === (5) * 8 === 40
```

The same expectations we hold for mathematical operators should also apply to pure functions as well. Now that you’ve caught a glimpse at the fundamental principles behind functional programming, perhaps its definition is more tractable now. So what is functional programming?

Functional programming refers to the declarative evaluation of pure functions to create immutable programs by avoiding externally observable side effects.

BECOMING FUNCTIONAL

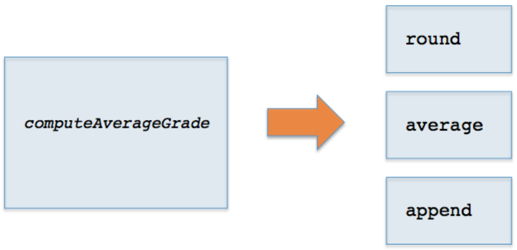
In order to become functional, you must learn to think functionally and have the proper tools to develop your *functional awareness*—the instinct of looking at problems as a combination of simple functions that together provide a whole solution. Functional programming is a software paradigm that, while based on very foundational concepts, will completely shift the way you think about a problem. When thinking about your application design, ask yourself the following questions:

- Extensible: Can I extend my code without having to refactor?
- Easy to modularize: If I change one file is another file affected?
- Reusable: Do you find yourself duplicating lots of code?
- Testable: I am having a hard time writing unit tests?
- Easy to reason about: Can I look at the code and visually follow what's happening?
- Malleable: Can I inject additional functionality without having to heavily rewrite my programs?

Let's explore the benefits functional programming brings to your JavaScript applications:

Modularizing your code

At a high level, functional programming is effectively the interplay between decomposition (breaking programs into small pieces) and composition (joining pieces to make programs). It is this duality that makes functional programs modular. As I mentioned previously, the unit of modularity, or "unit of work" is the function. Thinking functionally typically begins with decomposition by learning to break up a particular task into logical subtasks (functions):



If need be, these can be decomposed further until arriving at simple, pure functions, each of which is an independent unit of work. Modularization in functional programming is closely related to the *singularity* principle, which states that functions should have a single purpose, which is exactly what I was able to achieve.

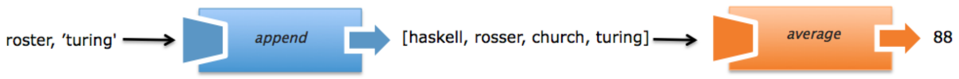
Functional composition can be understood simply as taking the output of one function and plugging it into the next. The result of composing two functions together is another function that gets called with the actual arguments:

```
f • g = f(g(x))
```

It reads: "*f* composed of *g*," which supposes a type-safe relationship between what *g* returns and *f* receives. As seen in our previous example:

```
var computeAverageGrade = compose(average, append);
```

Behind the scenes, the new roster array returned from `append` will be passed into `calcAverage`:



Given a class roster array with students Haskell, Rosser, and Church with grades 97, 82, 100, respectively, if Alan Turing's grade in the class was a 75, the new average will be:

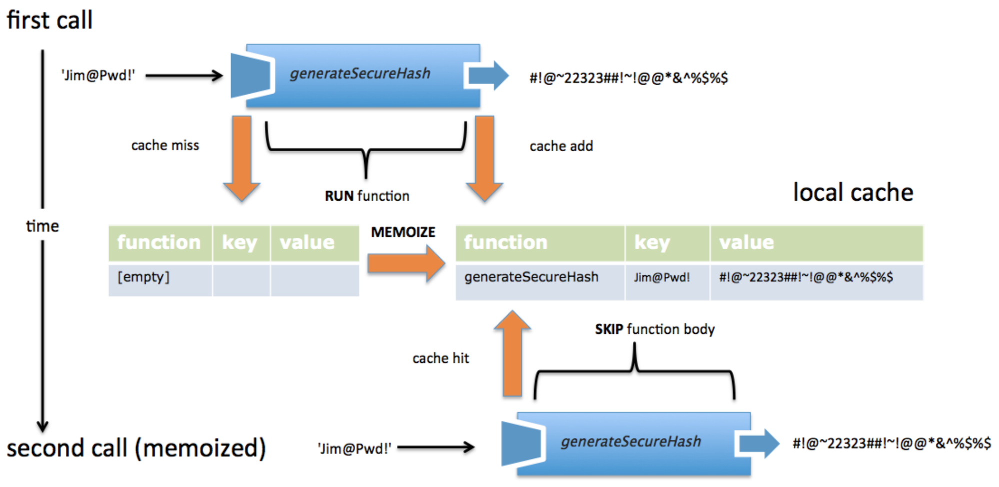
```
computeAverageGrade(roster, 'turing'); //-> 88 (better luck next time
Alan)
```

The concept of pure functions transitively applies to `computeAverageGrade`. Because it was built from the composition of two pure functions, it is itself pure. Understanding `compose` is crucial for learning how to implementing modularity and reusability in functional applications, as it leads to code where the meaning of the entire expression can be understood from the meaning of its individual pieces, a quality that becomes very hard to achieve in other paradigms. Also, it raises the level of abstraction so that you can clearly outline all of the steps performed in this code without being exposed to any of its underlying details. Because the `compose` function accepts other functions as arguments, it is known as a high-order function, a artifact in functional languages.

Functional programming not only makes your code more modular, but in some cases it can also speed up the evaluation of your functions.

Optimizing your function calls

Functional programming will not speed up the evaluation times of your individual functions; rather, its strategy is based on avoiding duplicated function calls, which can potentially speed up your application overall. This technique is called *memoization,* which originates from the premise of functions being referentially transparent. Recall that referentially transparent functions always return the same value on same inputs. Through memoization the set of input parameters are encoded as a string or number and used as the cache key to a function's result, internally persisted and looked up when the function is invoked again on the same input.



Consider a function that computes a secure hash based on a user's password. Depending on the type of algorithm used, this function can be very expensive to call. Generally, with secure encryption the more intense the algorithm the stronger the hash will be. Caching these results can be a tremendous improvement.

```
function generateSecureHash(password) {
     // run expensive algorithm

     return hash;
}
generateSecureHash('Jim@Pwd!'); //-> #!@~22323##!~!@@*&^%$%$
```

Calling `generateSecureHash` again on this same input bypasses the computation and returns the memoized result. The effectiveness of memoization is directly proportional to the level modularity in your code. As you decompose big tasks into a greater number of smaller, single-purpose functions, memoization can be used more efficiently to achieve a higher level of fine-grained caching and optimization.

Memoization is not a native JavaScript feature, but can be installed with API code by augmenting JavaScript's extensible `Function` object.

The beauty of this level of caching in functional languages like JavaScript is that it can be implemented without adding any boilerplate code of placing cache checks and inserts all over your functions.

While on the topic of writing boilerplate code, consider null checks. Handling errors and null values generally leads to an explosion of if-else conditions in your code. This excessive use of branching logic creates code that is hard to maintain and test. As an example, suppose I need to extract the country name of a student's school address. I think we've all seen similar convoluted code before:

```
function getCountry(student) {
   var school = student.getSchool();
    if(school !== null) {
           var addr = school.getAddress();
           if(addr !== null) {
              var country = addr.getCountry();
              return country;
           }
           return null;
     }
     throw Exception('Unknown');
 }
```

This anti-pattern of clean design is addressed with functional data types called monads used for null-safe operations as well as high-level abstractions that will allow you to write more robust, extensible code while handling errors effectively.

Handling errors fluently using Monads

Functional programming places a clear distinction between pure error handling and exception handling. Our goal is to implement pure error handling as much as possible, and allow exceptions to fire in truly exceptional conditions, just as the ones described earlier.

I will use a purely functional data type called monads to order to treat potentially hazardous state as safe, encapsulated values that use high-order functions to abstract out and consolidate error-handling logic and defensive null-checks. By returning consistent, predictable values, referential transparency is preserved across function calls and error-bearing expressions can easily be combined in a very fluent manner.

This is possible using the `Maybe` monad. `Maybe` is a data type that implements a very intuitive concept: a value is maybe there, maybe not. When wrapping a potentially unsafe value, `Maybe` either returns an object containing the value or an object containing "nothing." Instead of returning unsafe `null` values that then you'll have to check for, let's show `getCountry` based on `Maybe`.

```
var getCountry = function(student) {
   return Maybe.of(student)
         .map('getSchool')
         .map('getAddress')
         .map('getCountry')
           .getOrElseThrow('Unknown Country!');
};
```

Building extensible code with function chains

Let's tackle a different problem. Suppose you are tasked to write a program that would compute the average grade of students that have enrolled in more than one class. Approaching this problem from a functional mindset, you can identify three major steps:

- Selecting the proper set of students (whose enrollment is greater than one)
- Extracting their grades
- Calculating their average grade

```
_.chain(roster).filter( (student) => student.enrolled > 1)
               .pluck('grade')
               .average()
                   .value();
```

So far you've seen how functional programming can help us create modular, testable, and extensible applications. So, how does all this fare when running in a browser environment?

Streamline client-side development

Client side events come in many flavors: mouse clicks, text field changes, focus changes, etc. Consider the scenario of handling value changes on a text field and using the value entered to update a welcome banner. Naturally, as any JavaScript developer would do, I resort to jQuery:

```
$('#student-name').keyup(function(event) {
    var val = $(this).val();
    if(val == null || val == undefined || val.length == 0) {
        val = 'Anonymous';
    }
    $('#banner').text('Welcome: ' + val);
})
```

Arguably, this code is not very modular, as I am required to implement the logic of dealing with user input and updating the banner into a single function. Rather, the functional approach to handling events is to subscribe to asynchronous streams and, in a similar fashion to traversing an infinite collection of items, be able to compose a series of functions onto any event instance in a way that is separated from the actual browser event handling logic. In addition, I want to be able to port the concepts of referential transparency and immutability. Client side developing in functional is done reactive with *Functional Reactive Programming* (FRP).

FRP is the combination of reactive programming, or programming with data streams, with a strong focus on composition. For this I can leverage a functional library called Bacon.js that enhances JQuery's API to define the `asEventStream` function that works with asynchronous data streams. For example, to subscribe to all `keyup` events on an input field, I can write the following code:

```
var getValue = function(event) {
    return $(event.target).val();
};

var cleanValue = function(val) {
    return isNull(val) ? 'Anonymous' : val;
};

var processTargetObject = compose(cleanValue, getValue);
```

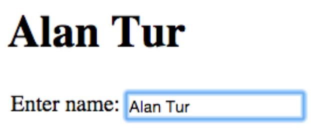
This function can be mapped onto a `studentName` stream:

```
var studentName = $('#student-name').asEventStream('keyup')
        .map(processTargetObject)
```

```
              .toProperty();
```

Here’s where the functional programming comes in. The `onKeyUp` function is like an event bus that can treat events as if they were elements in a collection, which means I can easily map functions onto it to immutably manipulate the target objects. I will create a function to extract and run logic on the value in the event stream. Calling the `toProperty` method on the stream creates a value accessor on any `keyup` events that occur on that HTML field, which I can listen for and handle within the `onValue` method. This allows me to cleanly separate the logic of extracting the target value from updating the banner message:

```
studentName.onValue(function(val) {
    $('#banner').text(val);
});
```


Alan Tur

Enter name: Alan Tur

Now, I pose to you the same questions with which we started the section: Is this program: Modular? Testable? Extensible? Readable? Maintainable? Functional programming is a paradigm shift that can drastically transform the way in which you tackle solutions to any programming challenges. So, is functional programming a replacement for the more popular object-oriented design? Fortunately, applying functional programming to your code is not an “all or nothing”.

In fact, lots of applications can benefit from it when used alongside an object-oriented architecture. Due to rigid control for immutability and shared state, functional programming is also known for making multi-threaded programming more straightforward. Because JavaScript is a single-threaded platform, this is not something we’ll need to worry about or cover in this book.

In sum, client-side applications are becoming more and more complex every day and, if done poorly, can become a nightmare for companies to maintain. Some companies actually enforce the use of certain paradigms and coding styles, others don’t and welcome the investment of time to learn new and better ways to accomplish certain tasks. Hybrid languages like JavaScript become really useful in these cases as they allow developers to experiment with a mix of styles. Whichever your situation is, I truly believe getting familiar with functional programming, even if you had no immediate plans to move into it, can truly enhance your skills and your ability to tackle complex problems in a way not possible using conventional methods.