Udemy Notes: Functional Programming in JS

Sean al-Baroudi

December 24, 2021

1 Introduction to Functional Programming[1]:

- **Def:** Functional Programming: the process of building software by composing pure functions, avoiding shared state, mutable data and side effects. This process is declarative rather than imperative. Application state flows through pure functions.
- **Avoids:** Side Effects, Mutations, Shared State.
- **Uses:** Pure Functions, Functional Composition, Declarative Coding Styles.
-

2 Side Effects and Pure Functions:

2.1 Tutorial:

- A function without side-effects is easy to reason about: you change this function and other sections of code will not break.
- **Def: Side-Effects:** An un-intended change of state outside the scope of a given block of code.
- **Def: Pure Function:** A function which entirely depends on the input passed to it - not on external data outside of its immediate scope.
- **Implications of this Definition:**
  - ☐ This means changes to state outside of the functions scope will have no effect on the function itself.
  - ☐ Likewise, changes to the internals of the function will have no effect on outside state.
  - ☐ Given the same input, the function will **always** produce the same output.
- **Kinds of Side Effects:**
  1. Changing a global value.
  2. Changing the original value of a functions argument.
  3. Throwing an exception.
  4. Printing to the screen or debugging.
  5. Triggering an external process.
  6. Invoke other functions that are not pure.
  7. Parameters passed by reference.
- **Note:** You can't build an application with no side effects, as *interacting with the user* is a form of side-effect. FP Practices try to *minimize and manage side-effects*, and their consequences.
- **Benefits of FP:**
  - ☐ Minimization of side-effects.
  - ☐ Cleaner, more modular code.
  - ☐ Code that is easier to debug.
-

Exercise Notes:

- In this tutorial we have functions that cause side-effects, and try to write pure functions.
- At this point of the course, we do not possess any methods to avoid Javascript's pass by reference mechanism. We cannot make copies of objects.
- The solution I wrote was written as if pass by local copy was implemented.
- It is important to note that the Instructor has done the same thing - this is just an exercise to get us thinking. It **cannot be solved** with our toolset at this point.
- I made the following changes:
  1. Removed CurrentUser and returnUser methods.
  2. Made a getUser function, that accessed a specific record and returned it.
  3. Made the score and tries functions pure.
  4. Made an updateUser function that accesses global scope to directly change the users array. However, this is the **only point of access** (controlled interface).
  5.
-

3 Avoiding Shared State and Mutations:

3.1 Shared-State and Mutations:

- **Def:** Statefulness: A program is considered stateful if it is designed to remember data from events or user interactions. This recoreded information is called the *"state of the program"*.
- In Javascript, state is stored in variables, constants and objects. At any given time, their allotments and contents define the state of hte JavaScript instance.
- **Def**: Shared State: (i) Any variable, object or memory space that exists in a shared memory scope. (ii) An object property where it is passed between scopes (like with function calls).
- Why it is "shared" for (ii)? Because objects are passed by reference by default in JS, and are mutable.
- **Recall: FP Definition** Application state is passed *through pure functions*.
- **Avoiding Mutations:**
- *Const keyword:* note that this simply freezes the stored content of data (makes it immutable)
  - ☐ Caveat: Objects: the pointer to the object is frozen, but the contents of the object are still mutable!
- **Solution: Use Object.freeze()** method on the object - all fields will be made mutable.
- Defining an object instance directly: Use the following template code:

```
let obj = {
  field1: "Bill",
  field2: "Carson",
  subObject: {
    subfield1: {success: true, value:1},
    subfield2: {success: true, value:34}
  }
};
```

  Each pairing of {} indicates another sub-level of object to be accessed. Each ":" indicates another field ⇔ value pair.
- **Cloning Objects:** To pass state through our program, and avoid sharing state with objects, we can use a few different methods to clone objects.
- Cloning the objects makes them mutably indepenent, which is what we want.
- Assign Method: To do a shallow (one level) independent cloning, use the following:

```
let obj2 = Object.assign({}, obj)
```

  Note that the brackets passed in here denotes the empty object.
- Caveat: The assign method will not do a deep copy. If you have sub-objects, these still reference the original sub-objects. Copy is only one level deep.
- Deep Copy Method: Use the following code:

```
let obj3 = JSON.parse(JSON.stringify(obj));
```

  Note here that the inner function convets the object to a JSON string, and the outer function interprets it, and builds a JS Object!
- Arrays of Objects: The same methods and effects for Object containers also apply to Arrays of Objects.
- Encoding of JS Objects: Is done via JSON string notation. Keep this in mind!
- **Spread Operation (...):** Is another way to produce a shallow copy for objects.

```
const newArray = [...oldArray];

const newObj = {...oldObj};
```

-

3.2 Reduce, Map and Filter Operations:

- Reduce, Map and Filter (RMF), allow us to use functions to perform looping operations.
- These can be used on any array.
- Each of RMF is a **higher order function** - it takes in a function as an argument.
- **Def: Reduce:** reduce() and reduceRight(): combines the elements of an array using the function you specify.

$$Reduce : f, A \rightarrow f(A)$$

where f(A) is a scalar value or singular output.
Note that reduce() starts from the Left of the array (start).
- **Def: Map:** Passes each element of the array to the function provided, and returns a new array of values that were operated on.

$$Map : f, A \rightarrow [f(1), ..., f(n)]$$

Anonymous function requirements: *Must return f(a_i)* for every element of the array.
- **Def: Filter:** Takes in an array and function, and returns a subset of the array.

$$Filter : f, A \rightarrow A'$$

Where $|A| >= |A'|$.
- **Caveat:** Applying RMF to arrays does not modify the arrays - *deep copies are performed* in these functions already.
- The funciton that we feed into these methods is an anonymous template function, that has the following header for Map and Filter:

```
function(val,index,array) {
  //Body
};
```

val is the selected value from the array.

index is its position in the array (optional)

array is the pointer to the array itself.
- An example of usage for these functions is below:

```
let newArray = arr.map(function(val) {
  return val ** 2;
});
```

Note: Map **must return f(a_i** after every call of the anonymous function.

At the end of all the calls on the Array, a new array is assembled with all of the f(a_i) results.

```
let filterArray = arr.filter(function(val) {
  return (val < 3);
});
```

Note: The anonymous function for filter **must return a true or false value**.

These values are compiled at the end, and used to pick out element of the original array.
- Note that reduce **is special**, and has an additional construct that can be referenced for its anonymous header:

```
function(accumulator,val,index,array) {
  //Body
};
```

An example of an accumulator usage is below:

```
let total = arr.reduce(function(accumulator, val) {
  return accumulator + val;
}, 0);
```

Here, accumulator is just a summation variable, that is passed from one call to the next. Note our initialization at the end of the call.

Note: You can bypass the accumulator - it doesn't have to be used. You can also just have a function that selects one element and returns it, Such as with the code below:

```
ar getUser = function(arr, name) {
    return arr.reduce(function(obj, val) {
        if (val.name.toLowerCase() === name.toLowerCase()) {
            return val;
        } else {
            return obj;
        }
    }, null);
```

-

4 Functional Composition:

4.1 Review: Higher Order Functions and Closures:

- **First-Class Functions:** A function can be used and stored anywhere a value can.
- Practically, this means that we can store functions in variables, and return functions from other functions.
- **Examples of First Class function usage:**
- Storing a function in a variable:

```
let myFun = function() {
//Body
};
```

- Storing a function in an array, and calling it later.

```
let myarr = [1,2,function() {\\Body};)];

\\To Call it.
myarr[2]();
```

- Storing a function in an object:

```
let obj = {
property1: 5,
funName1: function() {//Body;}
};

\\To Call it:
obj.funName1();
```

- Calling a function in a console statement:

```
console.log(28 + (function() {return 10;})());
```

Note the awkward Parens and Braces at the end!
- Returning a function from another function:

```
var returnFun = function() {
  return function(){console.log("Inner function says hello!")};
};
```

- Passing a function as an argument to another function:

```
var addTwo = function(num, fn) {
    console.log(num + fn());
};

addTwo(28, function() {return 15;})
```

- **Def: Higher Order Functions:** Functions that can take other functions as arguments, or return functions. In essence, functions that can operate on other functions.
- **Corollary:** The fact that all (i) all JS functions process and returns values, and (ii) Functions are first class, necessitates higher-order functionality.

4.2 Scope and Closure:

- **Def: Scope:** Rules that determine where within our program, variables and functions are accessaible.
- Javascript uses global and functional scope, for running programs.
- **Def: Closure:** The local variables of a function, kept alive after the function has returned. A closure practically occurs when a child funciton still has access to its parent function scope, even after the parent function has returned.
- **Examples of Closures:**
- setTimeOut() for inner functions, after parent has returned.
- Inner EventListeners set in a parent function, and a parent returns. Practical Code Example:

```
\\For HTML5 Button on a webpage:

var counter = function counter() {
var cnt = 0,
item1 = document.querySelector("#item1")

item1.addEventListener("click", function count() {
cnt++;
print();
});
};
```

- **Lingo:** We say that the inner or child function has created a **closure over** the parent scope. As it has not terminated and references the parent's variable scope (memory), JS must keep the parent scope around until child termination occurs.

4.3 Function Composition:

- **Def:** Function Composition: the purpose of chaining functions together to produce a new function, or accomplish some computation.
- **Functions vs Procdures:**
- Functions have an input, return a value, and are simplified to a single task.
- Procedures (typically): Encompass a block of code that accomplishes some set of tasks task; arguments and return values may or may not be specified, shared state and global scope may be used. *Many statements are often blocked together to accomplish some goal.*
- **In short:** Procedures are usually not simple to understand, functionally pure, and suffer from side-effects.
- **Arrow Functions:** A short-hand notation for creating anonymous/inline functions quickly. There are quite a few notational quirks, so see below:

```
\\2 Argument:
var sum = (a,b) => a+b;

\\One Argument:
var exp = n => n*n

\\No Arguments:

var constFun = () => 35;
```

Note: If you need to have one or more statement in the body, you need to encase these in braces {}.

- **Types of Function Composition:**
  1. Traditional (Nested) Calls: Function calls are nested within one another, like so:

     $$f_n(f_{n-1}(...f(x)))$$

     The call begins from the most nested function, and each output is fed into the next function's input.

     The problem with this method is that the calls are unwieldy, and read backwards.
  2. **Pipe Method:** We provide a list of functions, and setup a closure. We then hand the functional call setup a starter value, and it executes the chain of functions

```
const pipe = function(...fns) {
    return function(x) {
        return fns.reduce(function(v, f) {
            return f(v);
        }, x);
    }
};

Usage: pipe(fn1,fn2,fn3...)
```

  3. Compose Method: Same as Pipe, however reduceRight() is used, and arguments are reversed for the chain.
-

Arity and Currying:

- **Def:** Partial Application: (i) a function which has been applied to some of its arguments (but not all of them). It is a function which has some argument fixed inside its closure scope.

  Mathematically, the transformation of mapping can be viewed as:

  $$f : A \times B \times C \to D \Longrightarrow (a, b)_{fixed} \times C \to D$$

- **Def:** Arity: The number of inputs a function accepts. A function that accepts one argument is called an **unary** function.
- We have an issue with our functional composition methods: They only work if all functions are unary functions. If we have multiple arguments, or worse - functions with different arities, we cannot chain them together.
- We solve this with **currying, or "partial application"**
- In JS, we have a way to "bind" infomration to a function, to reduce all headers to just one argument!
- Bind Usage: Formally, bind is called as follows:

  ```
  functionname.bind(thisArg, arg1, arg2, ...)
  ```

  where thisArg allows us to explicitly set the this pointer (can be set to null), and the trailing arugments are bound to the *functionname*'s scope.

  An practical example is below:

  ```
  const partGetUser = getUser.bind(null, users);
  ```

- **Def: Currying:** The process of converting a N-ary function into a sequence of unary functions that feed into one another.

  Mathematically, the transformation of mapping can be viewed as:

  $$f : A \times B \times C \to D \Longrightarrow (A \to (B \to (C \to D)))$$

- The function itself after being curried returns the same values for all points on the domain - even if its form is different.
- The final function in the currying chain returns a value, not another function.
- Advantages of Currying:
  1. Can be used to specialize functions.
  2. Simplifies functional composition by reducing everything to unary functions.
-

5 Understanding Declarative Programming:

5.1

- **Def:** Imperative Programming: telling the computer how to accomplish some task.
-
- **Def:** Declarative Programming: expresses the logic of a program without identifying the control flow. Control flow is abstracted away, so declaritive code only needs to specify what to do.
- **What does this actually mean?** We have generic functions that perform the role of usual control logic (IF/LOOP, etc), and we ideally follow a functional paradigm to pass state through pure functions, to achieve some result.
- Note that we have already replaced some control flow statements (RMF) - we no longer use FOR loops.
- Now we are in a position to understand Eric Elliot's Functional Programming Definition:

- *"Functional programming is the process of building software by (i) composing pure functions, (ii) avoiding shared state, mutable data, and side-effects. Functional Programming is (iii) declaritive rather than imperative, and (iv) application state flows through pure functions".*
- **Some Implications and Clarifications:**
- (i) functions that depend entirely on the data passed to them, and return data without mutating data outside of them.
- (ii) With these effects, debugging and upgrading software becomes harder.
- (iii) Control structures and "how to" code, is removed. Our logic and control flow are all expressed through pure functions.
- (iv) Our application state flows through functions - and is tightly protected from side-effects.
- **How v.s What distinction:** How does Functional Programming specify *"what"* more than *"how"*? This is done through the creation of our pure and generalized functions - where their goals are clearly defined by their names. For example, with looping, we don't need to manually iterate indexes and make the computer tabulate things. We use a MAP function to apply a function to each element - which is what **we actually wanted to do.**
- **Caveat:** Functional programming's common support functions (RMF), are still **imperative!** We have chosen to genrealize the "how", so we can be more abstract and declarative.

6 Functional Programming Example: Compare OOP to Functional:

- Object Oriented Approach: In this example, we saw that user information and funcitons were siloed in object prototypes. Everything revolves around objects, and methods we can apply to them. The code was quite bloated and awkward.
- Functional Approach: Here, we had many pure helper functions and a few larger functions to perform the behaviours that we wanted. The functions are the center of our design, and data flows through them.
- **Function Approach Design Methods:**
  1. Write out/Pseudocode the large tasks you wish to accomplish.
  2. Break up the tasks into smaller tasks, idenitfying pure functions and reusable pieces of code.
  3. Write the smaller functions. *Give them Intention Revealing Names.*
  4. Write composite or larger functions that unite the smaller functions, to perform the application logic.
-

7 Lodash:

- Lodash is a JS framework that includes a functional programming paradigm.
- refer to the Lodash site to install the NPM package, or use the CDN link to load in browser without install.
- It expands the basic FP paradigm that comes with JS EcmaScript 6.
- It auto-curries partially applied functions. So for an N-a
- All FP functions we care about are accessed from the top level library. We access this using the reserved underscore operator.
- Some Examples:

```
const lodashRZS = _.filter(val => val > 0);

const calcAverage = (arr) => _.mean(arr);
```

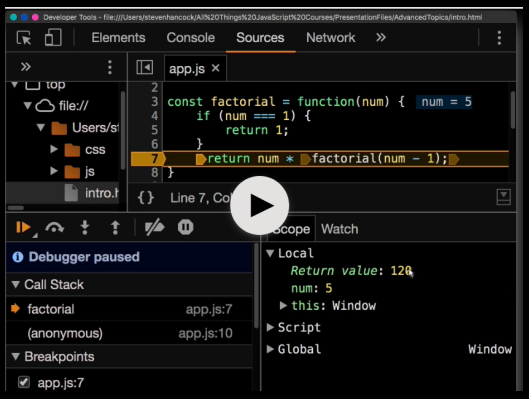
- Pipe and Compose are named "flow" and "flowRight", respectively.
-

8 Ramda:

- For Ramda, the notation is the same as Lodash. Replace the underscore with 'R".
- Ramda is less general, and primarily focuses on Functional Programming. There seems to be some Lisp/Racket/SICP influence, as seen on there website.
- As usual, use NPM or CDN links to run the library in browser.
- Like Lodash, RMF and other funcitonal methods are not object functions embedded in the array class - these are standalone functions that take arrays as arguments! We need this for our curried pipelines.
-

9 Other Functional Techniques and Articles + Bonus:

- This section goes into a primer on recursion. This can be studied another time.
- I already have experience with recursion - so no notes are written.
- **Note!**. Console in Firefox/Chrome has watch values and a call stack you can examine. This helps for chains of functions and recursion. Don't forget to debug with this!



10 Remainder Topics (for Later):

- **Understand the Pipe Method:**

```
const pipe = function(...fns) {
    return function(x) {
        return fns.reduce(function(v, f) {
            return f(v);
        }, x);
    }
};
```

  - ▽ Explain the Closures that are used.
  - ▽ What is the middle anonymous funciton for? What is "x" here?
  - ▽ Provide a modified function to show understanding of its inner workings.
- **Understand the Currying Method:**

```
function curry(fn,arity = fn.length) {
    return (function nextCurried(prevArgs){
        return function curried(nextArg){
            var args = [ ...prevArgs, nextArg ];
            if (args.length >= arity) {
                return fn( ...args );
            }
            else {
                return nextCurried( args );
            }
        };
    })( [] );
}
```

  - ▽ Map out the closures, and memory frames for each iteration.
  - ▽ Map out the setup, and execution of the chain of functions one at a time.

References

[1] https://www.udemy.com/course/functional-programming-in-javascript-