Javascript Functional Programming by Example

D. Renault

JDEV 2015

July 2015, v. 1.0

What is functional programming ?

Programming style based on two principles :

- **First-class citizenship** of functions :
  1. A function can be named,
  2. A function can be passed as the argument of another function,
  3. A function can be defined anywhere in the code.
  4. A function can be returned from another function,
  5. A function can be stored in any kind of data structure,
- **Purity** of functions :
  - Reject side effects and state,
  - Advocate immutability of data structures.

Already investigated at length : Lisp, Scheme, Haskell, OCaml, Scala, Clojure ...

Main idea : learn from existing constructions and techniques

Anonymous functions / Closures

Anonymous function

A function in Javascript can be defined in the following way :

```
function (param1, param2, ...) { body };
   (param1, param2, ...)  ⇒ { body };
```

(ECMAScript 6)

```
var plus = function (x,y) { return x + y };
plus(4,5) // → 9
```

- An anonymous function can be defined anywhere in the code.
- Functions can be seen as code chunks, facilitating abstraction and reuse.

Closure

When a function is defined, the variables it uses that are not parameters are embarked (by reference) within the function as an immutable record.

```
function createClosure() {
  var noDirectAccess = 'secret';
  return function () {
    console.log(noDirectAccess)}}
```

```
var showSecret = createClosure();
// noDirectAccess is not defined
// but showSecret can still reach it
showSecret(); // → secret
```

An example of closure : the Module pattern

Intent

Create an encapsulating structure that can store an internal private state, and expose a public interface.

```
var myModule = (function () {
    var conf = {                              // Private to the function
        useCaching: true,
        language: 'en'
    };
    return {
        reportConfig: function () {
            console.log('Caching ' +
                        (conf.useCaching ? 'enabled' : 'disabled'));
        },
        updateCaching: function(caching) {
            conf.useCaching = caching; } // Restricted access to config
    }})();
```

- Reminiscent of Scheme emulation of objects with closures.
- Enables a modular programming style.

First-class citizenship : second rule (1/2)

Functions taking functions as parameters :

- Callbacks (cf. JQuery events) :

```
$.ajax({
    url:     '/api/getWeatherTemp',
    data:    { zipcode: 33333 },
    success: function( data ) {
        $('#weather-temp').html('<strong>' + data
                               + '</strong> degrees');
    }});
```

- Generic code via higher order functions :

```
function iterUntil(fun,valid){
  var result = undefined;
  while (!valid(result)) {
    result = fun(result); }
  return result; }
```

```
// Calls requestCredentials()
var res = iterateUntil(
  requestCredentials,
  _.negate(_.isUndefined)
);
```

Easier within a framework such as Underscore.js which provides a bunch of higher order functions : `each`, `map`, `reduce`, `filter` ...

First-class citizenship : second rule (2/2)

Functions taking functions as parameters :

- JsCheck is a **specification-based** testing tool based on the ideas of Quickcheck in Haskell, and developed by Crockford.

> Each function under test is associated to an abstract specification in the form of a set of predicates.

Function under test :	`passwordScore(password)` →score
Specification :	all passwords without special characters must have a negative score.

```
JSC.claim('Negative score for passwords w/o special characters',
   function (verdict, password, maxScore) {
    return verdict(passwordScore(password) < 0);
}, [JSC.one_of([
        JSC.string(JSC.integer(5, 20), JSC.character('a', 'z')),
        JSC.string(JSC.integer(5, 20), JSC.character('A', 'Z')),
    ])]);
```

Currying / Partial application

Currying

Process of transforming a function of multiple arguments into a sequence of functions each with a single argument.

```
function plus_plain(x,y) {
  return x+y;
}

plus_plain(4,5) // → 9
```

```
function plus_curry(x) {
  return function (y) {
    return x+y }
}
plus_curry(4)(5) // → 9
```

- Advantages : in the curried form, possibility to partially apply a function.
- Example with the `partial` function of Underscore.js :

```
var sendAjax = function (url, data, options) { /* ... */ }

var sendPost = _.partial(sendAjax,
                         _, _,          // '_' parameters stay
                         { type: 'POST', // other ones are fixed
                           contentType: 'application/json' });
```

⇒ Allows to specialize generic functions.

First-class citizenship : fourth rule

Functions returning functions :

- Smoothen the use of higher order functions :

```
function plucker(field) {
  return function(obj) {
    return (obj && obj[field]);
  };}
```

(cf. pluck in underscore.js)

```
var oldies = [
  {name: 'pim',  color: 'green'},
  {name: 'pam',  color: 'red'},
  {name: 'poum', color: 'blue'}];

_.map(oldies, plucker('name'));
// → ['pim','pam','poum']
```

- Functions as chunks of code : Underscore.js templates

```
var compiled = _.template("\
<% _.each(items,
          function(item,key,list){ %>\
          <tr>                        \
            <td><%= key+1 %></td>     \
            <td><%= item.name %></td>\
          </tr>                       \
    <% }) %>");
```

(compiled is a function of items)

```
compiled({items: oldies});
```

```
<tr><td>1</td>
    <td>pim</td></tr>
<tr><td>2</td>
    <td>pam</td></tr>
<tr><td>3</td>
    <td>poum</td></tr>
```

Function composition (1/2)

Natural way of manipulating functions ⇒ via composition.
Here composition appears as a composition of methods via the "·" operator.

- Write code in a declarative manner – Underscore.js `chain`

```
_.chain([1,2,3,200]) // Compose the following actions on this array
   .filter(function(num) { return num % 2 == 0; })
   .tap(alert)
   .map(function(num) { return num * num })
   .value();         // And return the result
```

- Compose abstract creation rules to create complex objects – AngularJS routes

```
$routeProvider // Compose rules for routing URLs
   .when('/', {
     controller:'ProjectListController as projectList',
     templateUrl:'list.html',
     resolve: {
       projects: function (projects) { return projects.fetch() }}})
   .when('/edit/:projectId', {
     controller:'EditProjectController as editProject',
     templateUrl:'detail.html' })
   .otherwise({
     redirectTo:'/' });})
```

Function composition (2/2)

- Extend behavior : functions can be decorated, and code can be added before, after and around the call of the function.

  Example : Underscore.js `wrap` function

```
User.prototype.basicLogin = function () { /* ... */ }

User.prototype.adminLogin =
  _.wrap(User.prototype.basicLogin,
            function (loginFun) {
    var hasAdminPrivs = this.admin;
    if (!hasAdminPrivs)
        console.log('!! Cannot login ' + this.name + ' as admin');
    else {
        loginFun.call(this);    // Call basicLogin here
        console.log(this.name + ' has logged in as admin');
    }});
```

  - Akin to Lisp method combinators, Python decorators, Rails method callbacks.
  - Allows to do **aspect-oriented programming** (cf. meld.js).

Data structures

Some data types compose well with functional programming.

- **Lists** :

  JQuery selectors mechanism is a way to represent set of DOM nodes.

```
$( 'li' ).filter( ':even' )
         .css( 'background-color', 'red' );
```

  Akin to the C# LINQ, Java Streams, or the `List` monad in Haskell.

```
from node in nodes where  (n => n.tag == 'li')
                   where  (n => n.index % 2 == 0)
                   select (n => n.BackgroundColor('red'));
```

Data structures : trees

Some data types compose well with functional programming.

- **Trees**
  A DOM tree can be manipulated via higher-order functions (cf. Crockford) :

```
function walk_tree(node, fun) {
    fun(node);
    var tmpnode = node.firstChild;
    while (tmpnode) {
        walk(tmpnode, fun);
        tmpnode = tmpnode.nextSibling; }};
```

  And its functional version :

```
function fold_tree(tree, fun, start) {
    var newstart = fun(start, tree);
    return _.reduce(tree.children,
                    function(cur, subtree) {
                        return fold_tree(subtree, fun, cur); },
                    newstart);}
```

Data-driven programming

Functional data-driven programming

Technique where the data itself controls the flow of the program and not the program logic. In functional programming, the data may be a set of functions.

```
var funs = [
    { name:'cool', deps:[], func:function () { console.log('cool') } },
    { name:'hot',  deps:[], func:function () { console.log('hot') } },
    { name:'temp', deps:['cool', 'hot'],
                   func:function (cool,hot,val) {
                          (val > 10) ? hot() : cool() }}];
```

```
var injector = function(key) {
    var fobj  = _.find(funs, _.matcher({name:key}));
    var fdeps = _.map(fobj.deps, injector);
    return function () {
        var fullargs = fdeps.concat(arguments);
        fobj.func.apply(fobj,fullargs); }}
injector('temp')(12); // → "hot"
```

- Connections between functions handled by injectors.
- Angular JS dependency injection : https://docs.angularjs.org/guide/di

Control of execution

Considering computations in the code, several strategies are available :

- Call by value : evaluate every call at the point of definition,
- Call by need : leave the functions unevaluated until needed.

Allows some control on the flow of execution.

- **Lazy programming** – Lazy.js : http://danieltao.com/lazy.js/

```
var lazySequence = Lazy(array)
  .filter(_.matcher({ category : 'cat' }))
  .take(20);
  .map(template)
```

  - Evaluation is delayed until needed ⇒ no intermediary arrays;
  - Allows efficient operations on (possibly infinite) streams.

- Asynchronous Module Definitions – require.js : http://requirejs.org
  Control module dependencies to ascertain their loading in the correct order.

```
define(['dep1', 'dep2'], function (dep1, dep2) {
    return function () { /* ... */ }; //Define the module value
});
```

Memoization

Referential transparency

A pure function is referentially transparent : given the same parameters, it will always return the same results.

- **Caching results**, a simple form of lazy programming – Underscore.js `memoize`

```
Function.prototype.memoize = function () {
  var self = this, cache = {};
  return function( arg ){
    if(arg in cache) {
      console.log('Cache hit for ' + arg);
      return cache[arg];                // return result if in cache
    } else {
      console.log('Cache miss for ' + arg);
      return cache[arg] = self( arg );// apply function otherwise
}}}
```

  - Memoization can be done automatically.

And more to come . . .

Acceptance in the next ECMAScript 6 : http://es6-features.org

- Destructuring arguments – a form of **pattern-matching**

```
function logArray ([ head, ...tail ])   { console.log(head, tail) }
function logObject({ name: n, val: v }) { console.log(n, v) }
```

- Promises – a form of **continuation-passing-style** programming

```
getJSON('story.json').then(function(story) {
  addHtmlToPage(story.abstract);
}).catch(function(err) {
  addTextToPage('!! Error : ' + err.message);
}).then(function() {
  $('.spinner').style('none'); });
```

Purity

Pure function

A pure function in programming is a function in the mathematical sense, i.e. there is only one possible result for each possible arguments, and no other effect.

- Independence from context (do not read from external state),
- Referential transparency (invariant behavior),
- No side-effect (do not write to external state).

Consequences :

- Simpler unit testing,
- Easier parallelization of code (think map/reduce),
- Easier static checking.

Very limited support in Javascript (`const` variables, properties)

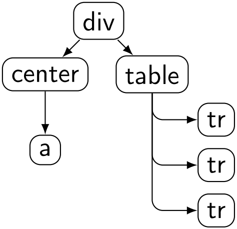
Immutable structures

Immutable-JS https://facebook.github.io/immutable-js/ :

- Provides several immutable data structures : List, Stack, Map, Set ...
- Maximises sharing and takes advantage and laziness for efficiency.

Example : `PureRenderMixin` in React.js using Immutable-JS structures.

→ render an HTML element if and only its components have been modified.




```
// react.js immutable structure
createElem("div",
   createElem("center",
      createElem("a")),
   createElem("table",
      createElem("tr"),
      createElem("tr"),
      createElem("tr"))
);
```

When rendering, the `render` function is called only if the element has changed.

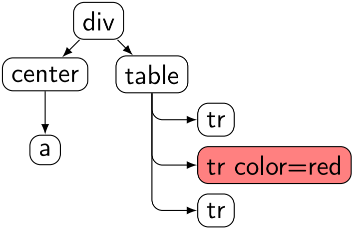
Immutable structures

Immutable-JS https://facebook.github.io/immutable-js/ :

- Provides several immutable data structures : List, Stack, Map, Set . . .
- Maximises sharing and takes advantage and laziness for efficiency.

Example : `PureRenderMixin` in React.js using Immutable-JS structures.

→ render an HTML element if and only its components have been modified.




```
// react.js immutable structure
createElem("div",         // render
   createElem("center",   // --
      createElem("a")),   // --
   createElem("table",    // render
      createElem("tr"),   // --
      createElem("tr"),   // render
      createElem("tr"))   // --
);
```

When rendering, the `render` function is called only if the element has changed.

Caveats

Lack of complete static checking hinders functional programming.

- Pure Javascript tools have a limited scope :
  - Crockford JSLint : avoid anonymous functions within a loop
  - Google Closure compiler : calling a non-function variable, wrong arguments count
- Tendency to evolve towards compilers to Javascript
  - Facebook Flow, Microsoft TypeScript, LLVM Emscripten ...

Most wanted missing features :

- Static (optional) type checking, with genericity for higher-order functions.
- Static verification at the modular level (or interfaces).

Good reading

- *Javascript : the Good Parts*,
  D. Crockford, O'Reilly Media, 2008
- *Functional Javascript*,
  M. Fogus, O'Reilly Media, 2013