Thinking in Functions

Improve Your Javascript Through Functional Programming Techniques

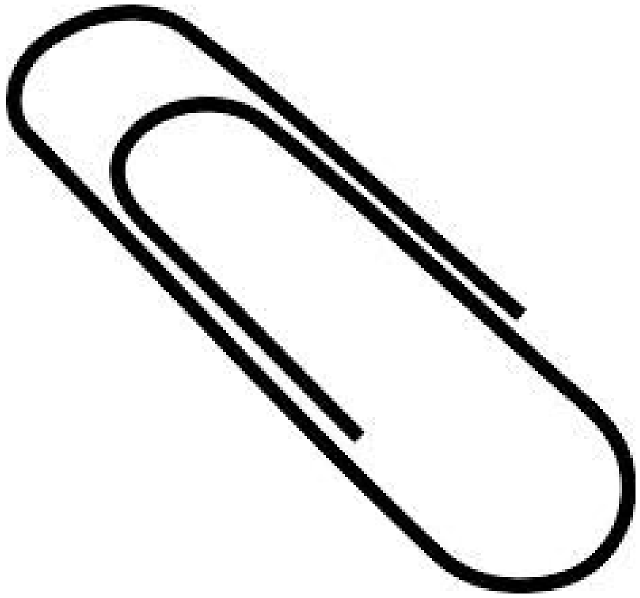
By - Anupam Jain

Software Design

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.

- C.A.R. Hoare, 1980 ACM Turing Award Lecture

Good Design



Good Design??

Two Equivalent Programs

```
var total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
```

```
total = sum(range(1, 10));
```

Which is more likely to contain a bug?

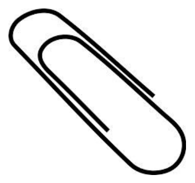
Abstraction

The “shared” vocabulary that allows us to write concise, high level, programs

Shared Vocabulary

```
var total = 0, count = 1;
while (count <= 10) {
  total = total + count;
  count = count + 1;
}
```

```
total = sum(range(1, 10));
```



Add the elements of an array

```
var array = [1,2,3,4];
var sum = 0;

for(var i=0; i<array.length; i++) {
  sum = sum + array[i];
}

console.log(sum);
```

Multiply the elements of an array

```
var array = [1,2,3,4];
var product = 1;

for(var i=0; i<array.length; i++) {
  product = product * array[i];
}

console.log(product);
```

Sum

```
var array = [1,2,3,4];
var result = 0;

for(var i=0; i<array.length; i++) {
  result = result + array[i];
}

console.log(result);
```

Product

```
var array = [1,2,3,4];
var result = 1;

for(var i=0; i<array.length; i++) {
  result = result * array[i];
}

console.log(result);
```

Abstraction

```
Array.prototype.reduce = function(operation, value) {
  var result = value;
  for(var i=0; i<this.length; i++) {
    result = operation(result, this[i]);
  }
}
```

Sum

```
function plus(a,b) {
  return a+b;
}

var sum = array.reduce(plus, 0);
```

Product

```
function mult(a,b) {
  return a*b;
}

var sum = array.reduce(mult, 1);
```

Why?

Is any boolean true?

```
function or(a,b) {
  return a || b;
}

var anytrue = array.reduce(or, false);
```

Are all booleans true?

```
function and(a,b) {
  return a && b;
}

var alltrue = array.reduce(and, true);
```

Are all elements larger than 3?

```
function check(prev,elem) {
  return prev && (elem > 3);
}

var allsatisfy = array.reduce(check, true);
```

Are all elements digits?

```
function check(prev,elem) {
  return prev && (elem >= '0' && elem <= '9');
}

var allsatisfy = array.reduce(check, true);
```

Do we have another pattern emerging?

More Abstraction!

```
Array.prototype.reduceMap= function(operation, value,
modify) {
  var result = value;
  for(var i=0; i<array.length; i++) {
    result = operation(result, modify(array[i]));
  }
}
```

Are all elements larger than 3?

```
function and(a,b) {
  return a && b;
}

function islarger(elem) {
  return elem > 3;
}

var alllarger = array.reduceMap(and, false, islarger);
```

Are all elements digits?

```
function and(a,b) {
  return a && b;
}

function isdigit(elem) {
  return elem >= '0' && elem <= '9';
}

var alldigits = array.reduceMap(and, false, isdigit);
```

Why

- Easy to understand what's going on
- Cleanly separated “logic” (`and` and `isdigit`)
- Common machinery (`reduceMap`) added to Array prototype to be easily reused
- Easy to test functions in isolation

But..

- `reduceMap` has 3 arguments that are not easy to remember
- It's harder to generalise this pattern. Should we keep on adding arguments to `reduce/reduceMap`?

We can abstract further

```
Array.prototype.reduceMap = function(operation,
value, modify) {
    ...
    result = operation(..., modify(...));
    ...
}
```

2 Arg Function Composition

```
Function.prototype.compose2 = function(func) {
  return function(a,b) {
    this(a, func(b));
  }.bind(this);
}
```

Refactored `reduceMap`

```
Array.prototype.reduceMap = function(operation,
value, modify) {
    ...
    result = (operation.compose2(modify))(..., ...);
    ...
}
```

Refactored `reduceMap` (contd.)

```
Array.prototype.reduceMap = function(operation,
value) {
    ...
    result = operation(..., ...);
    ...
}
```

When we set

```
operation = operation.compose2(modify))
```

But that's just `reduce` again!

Are all elements larger than 3?

```
function and(a,b) {
  return a && b;
}

function islarger(elem) {
  return elem > 3;
}

var alllarger = array.reduce(and.compose2(islarger),
false);
```

Are all elements digits?

```
function and(a,b) {
  return a && b;
}

function isdigit(elem) {
  return elem >= '0' && elem <= '9';
}

var alldigits = array.reduce(and.compose2(isdigit),
false);
```

Much Better

- We removed an extra function (`reduceMap`) which was actually not needed
- Our existing machinery (`reduce`) turned out to be powerful enough to support our logic
- We reduced code complexity by introducing another “orthogonal” and “generic” shared vocabulary (`compose2`)

Perfection

Perfection is attained not when there is nothing more to add, but when there is nothing more to remove

- Antoine de Saint Exupéry

We Can Go Deeper

- Generic Currying
- Generic Composition
- Generic “folds”

Compose & Curry

Curry

```
Function.prototype.curry = function() {
  return function(arg) {
    return this.bind(undefined, arg);
  }.bind(this);
}

function add(a,b) { return a+b; }

var add5 = add.curry()(5);
add5(10) // => 15
```

Flip Curry

```
Function.prototype.flip = function() {
  return function(a) {
    return function(b) {
      return this(b)(a);
    }.bind(this);
  }.bind(this);
};

function minus(a, b) { return a-b; }

var minus10 = minus.curry().flip()(10);
minus10(15) // => 5
```

Compose

```
Function.prototype.compose = function(f) {
  return function(a) {
    return this(f(a));
  }.bind(this);
}

var add15 = add.curry()(5).compose(add.curry()(10));
add15(10) // => 25
```

Are all elements digits?

```
array.reduce(and.curry().compose(isdigit).flip(),
false)
```

Because -

```
f.compose2(g) ====>  f.curry().compose(g).flip();
```

Thinking of functions as Data Transformers

```
===> f.curry().compose(g).flip()

f(a,b) = f(a, b)

f.curry(a)(b) = f(a, b)

f.curry().compose(g)(a)(b) = f(g(a), b)

f.curry().compose(g).flip()(a)(b) = f(g(b), a)
```

Point of Diminishing Returns

- There's a point when the abstracted approach becomes more cumbersome than the “native” approach
- Recognizing when that point happens will make you a better programmer

Recursive Look at reduce

```
Array.prototype.head = function() {  return this[0]; }


Array.prototype.tail = function() { return this.slice(1); }


Array.prototype.reduce = function(operation, value) {
  if(!this.length) return value;
  return operation(this.head(), this.tail().reduce(operation, 
value););
}
```

Any Recursive Data Structure Can Be “Reduced”

```
function Person(name, children) {
  this.name=name;
  this.children=children || [];
}

Person.prototype.reduce = function() …?
```

reducePerson

```
Person.prototype.reduce = function(personop, childrenop,
value) {
  return personop(this.name, reduceChildren(this.children));

  function reduceChildren(children) {
    if(!children.length) return value;
    return childrenop(children.head().reduce(...),
reduceChildren(personOpchildren.tail());
  }
}
```

Get names of all descendents

```
function append(val, array) {
  if(typeof val === 'string') val = [val];
  return [val].concat(array);
}

Person.prototype.descendents = function() {
  return this.reduce(append, append, []);
};
```

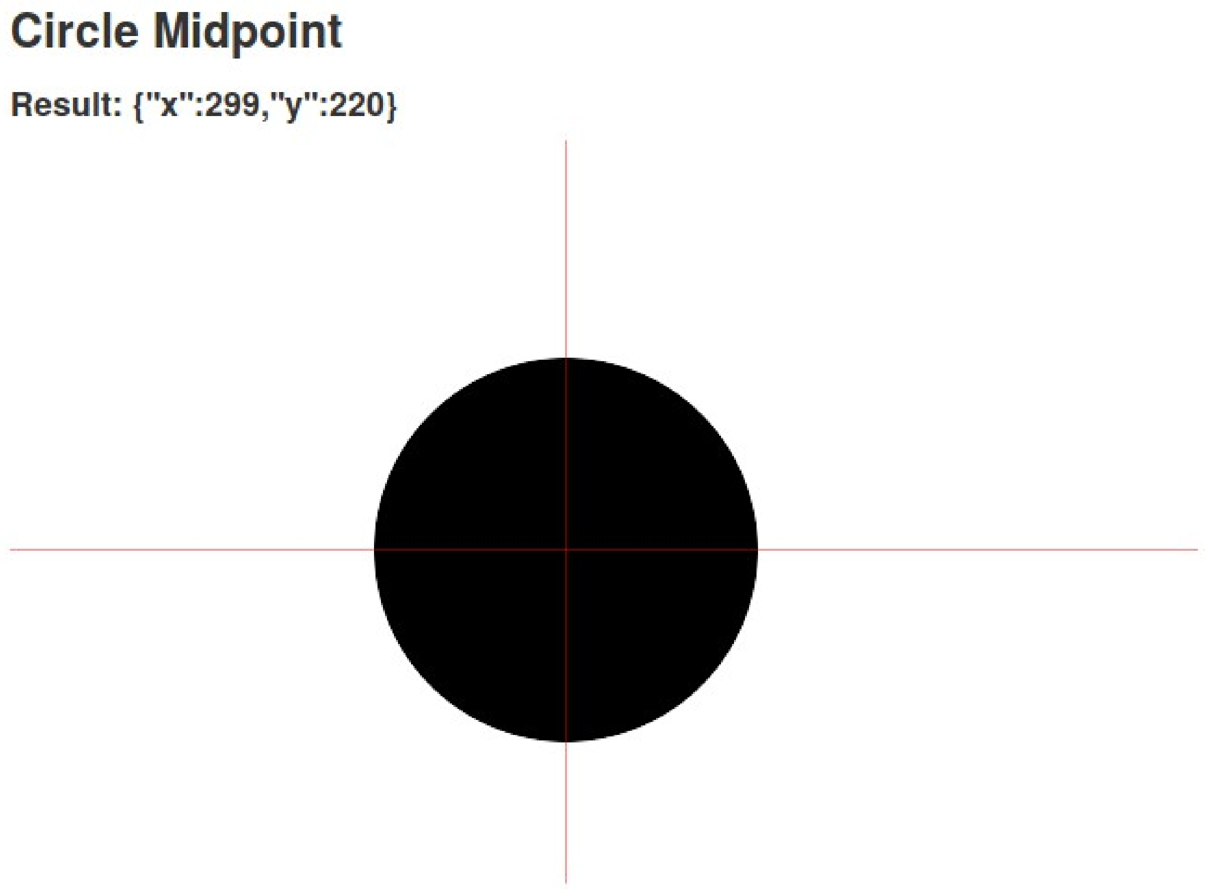
Uppercase all descendent names

```
function person(name, children) {
  return new Person(name, children);
}

function upper(str) {
  return str.toUpperCase();
}

Person.prototype.upperNames = function() {
  return this.reduce(person.compose(upper), append, []);
};
```

Another simple example of Abstraction


Circle Midpoint

```
Result: {"x":299,"y":220}
```

Find the midpoint

```
 // Get the midpoint of range
which satisfies the given predicate
function midpoint(minx, maxx, f) {
  for(x = minx; x < maxx; x++) {
    if(!f(x)) {
      mid = minx + x;
      if(mid % 2) mid = (mid-1)/2;
      else mid = mid/2;
      return mid;
    }
  }
}
```

```
// Find a point on the circle
loop:
 for(cx = 0; cx < width; cx++)
   for(cy = 0; cy < height; cy++)
     if(isBlack(cx, cy)) break loop;
```

```
// X coordinate
midpoint(cy, height,
isBlack.curry()(cx));

// Y coordinate
midpoint(cx, height,
isBlack.flip().curry()(cy));
```

Takeaways

- Abstract general higher order functions which can be combined with specialized “logic”
- Immutable Data helps Abstraction
- Think in terms of data being manipulated instead of “actors” that do the manipulation
- Compose simple functions together to build more complex logic

Thank You!

Questions?