Functional Programming in JavaScript

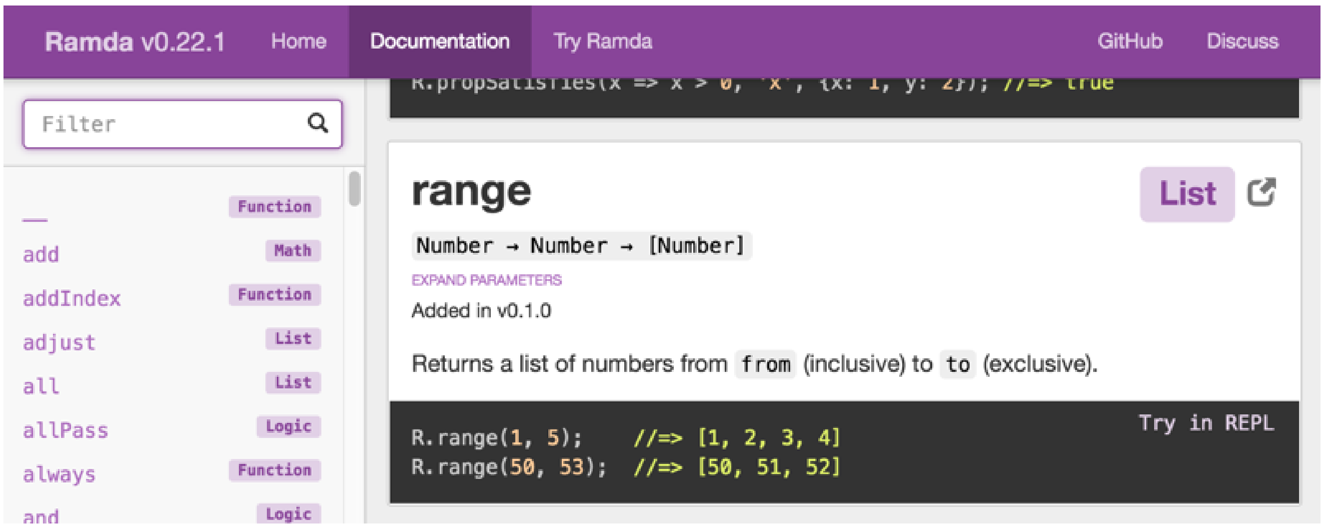
By Aaron Geisler and Sam Perez

Agenda

1. Ramda.js
2. Key concepts
3. Code examples
4. Functional programming at Social Native
5. Questions

Ramda.js

1. Immutability & purity
2. Curried functions
3. Consistent parameter ordering



Key Concepts

1. Immutability
    - Mutable objects
    - Side effects
    - Function purity
2. Currying
    - Function arity
    - Partial function application
    - Parameter reordering
3. Function composition
    - Functions of functions
    - Pipelines

Mutable Objects

'liable to change'

By default, js objects are mutable

```
let mutablePerson = {
  name: 'Jeff'
};

mutablePerson.name = 'Jeffrey';

console.log(mutablePeron.name); // => 'Jeffrey'
```

You can make an immutable object with some extra work

```
let immutablePerson = Object.freeze({
  name: 'Jeff'
});

immutablePerson.name = 'Jeffrey';

console.log(immutablePeron.name); // => 'Jeff'
```

Side Effects

'modifies some state or has an observable interaction with calling functions or the outside world'

Side effects come in many flavors:

1. Making HTTP calls
2. Writing to a database
3. Logging
4. Altering global state
5. Modifying pass-by-reference input parameters

Function Purity

A function is pure if...

1. Given the same inputs, it always evaluates to the same result
2. Does not have side effects

```
function impureFunction(person){
  person.name = person.name.toUpperCase();
  return person;
}

function anotherImpureFunction(person){
  if (Math.random() < 0.5){
    person.name = person.name.toUpperCase();
  }
  return person;
}

function pureFunction(person){
  return {
    name: person.name.toUpperCase()
  };
}
```

Immutability & Purity in Ramda.js

```
const a = ['write', 'more'];
const b = R.append('tests', a);

console.log(a === b); // => false
```

vs.

```
let a = ['write', 'more'];
a.push('tests');
```

Practical Benefits

Why write code this way?

1. Easy to test and verify correctness
2. Concurrency
3. Avoid bugs related to side effects and global state
4. You can 'trust' the functions you call

```
functionOfCompleteMystery(options); // Muhahaha
```

Function Arity

'the number of arguments a function takes'

```
function arityOfOne(x) {
  ...
}

function arityOfTwo(x, y) {
  ...
}

function arityOfThree(x, y, z) {
  ...
}
```

Partial Function Application

'apply arguments to a function producing another function of smaller arity'

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

const arityThree = R.curry(add);  // => function(a, b, c)

const arityTwo = arityThree(1);   // => function(b, c)

const arityOne = arityTwo(2);     // => function(c)

const result = arityOne(3);       // => 6
```

Parameter Reordering

```
function concat(a, b, c){
  return a.toString() + b.toString() + c.toString();
}

const arityThree = R.curry(concat);                    // => function(a, b, c)

const arityOne = arityThree(R.__, 2, 3);               // => function(a)

const result = arityOne(1);                            // => 123
```

Parameters in Ramda

Ramda is very consistent with parameter ordering:

- Objects, lists, and values are supplied last
- Functions are typically provided first
- Very easy to remember

```
R.map(mappingFunction, list);

R.reduce(reducingFunction, accum, list);

R.append(element, list);
```

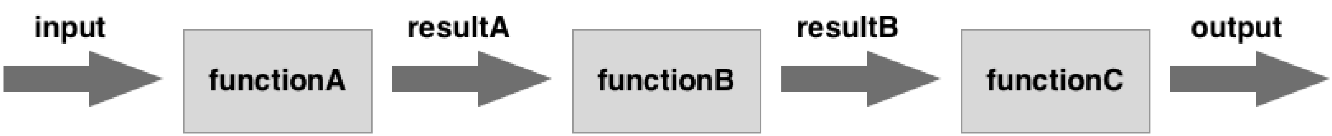
Function Composition

Similar to object composition in OOP - combine functions to create new functions.

We are going to talk about two:

1. Left-to-right function composition -> R.compose
2. Right-to-left function composition -> R.pipe

```
const output = R.pipe(
  functionA,
  functionB,
  functionC
)(input);
```



Example 1 - Nested Functions

Ever wrote something that looked like this?

```
function someFunction(x) {
  return foo(bar(baz(qux(x))));
}
```

R.compose to the rescue!

```
function someFunction(x) {
  return R.compose(
    foo,
    bar,
    baz,
    qux
  )(x);
}
```

Example 2 - Chained Thens

```
var getIncompleteTaskSummaries = function(membername) {
    return fetchData()
        .then(function(data) {
            return R.get('tasks', data)
        })
        .then(function(tasks) {
            return R.filter(function(task) {
                return R.propEq('username', membername, task)
            }, tasks)
         })
        .then(function(tasks) {
            return R.reject(function(task) {
                return R.propEq('complete', true, task);
            }, tasks)
        })
        .then(function(tasks) {
            return R.map(function(task) {
                return R.pick(['id', 'dueDate', 'title', 'priority'], task);
            }, tasks);
        })
        .then(function(abbreviatedTasks) {
            return R.sortBy(function(abbrTask) {
                return R.get('dueDate', abbrTask);
            }, abbreviatedTasks);
        });
};
```

Example 2 - Curried Functions

Before

```
var getIncompleteTaskSummaries = function(membername) {
    return fetchData()
        .then(function(data) {
            return R.get('tasks', data)
        })
        .then(function(tasks) {
            return R.filter(function(task) {
                return R.propEq('username', membername, task)
            }, tasks)
         })
         ...
```

After

```
var getIncompleteTaskSummaries = function(membername) {
    return fetchData()
        .then(R.get('tasks'))
        .then(R.filter(R.propEq('username', membername)))
        .then(R.reject(R.propEq('complete', true)))
        .then(R.map(R.pick(['id', 'dueDate', 'title', 'priority'])))
        .then(R.sortBy(R.get('dueDate')));
};
```

Example 2 - Pipe & Curry

```
const transformRecords = R.pipe(
  R.get('tasks'),
  R.filter(R.propEq('username', membername)),
  R.reject(R.propEq('complete', true)),
  R.map(R.pick(['id', 'dueDate', 'title', 'priority'])),
  R.sortBy(R.get('dueDate'))
);

const getIncompleteTaskSummaries = function(membername) {
    return fetchData().then(transformRecords);
};
```

Example 3 - Project Euler

The four adjacent digits in the 1000-digit number that have the greatest product are $9 \times 9 \times 8 \times 9 = 5832$.

```
73167176531330624919225119674426574742355349194934
96983520312774506326239578318016984801869478851843
85861560789112949495459501737958331952853208805511
12540698747158523863050715693290963295227443043557
66896648950445244523161731856403098711121722383113
62229893423380308135336276614282806444486645232...
```

Find the thirteen adjacent digits in the 1000-digit number that have the greatest product. What is the value of this product?

Example 3 - Imperative Approach

```
function largestAdjacentProduct(array, windowSize) {
  let len = array.length - windowSize;
  let largestProduct = 0;
  for (let i = 0; i < len; i++){
    let product = array[i];
    for (let j = 1; j++; j < windowSize){
      product *= array[j + i];
    }
    if (product > largestProduct){
      largestProduct = product;
    }
  }
  return largestProduct;
}
```

Example 3 - Declarative Approach

```
function largestAdjacentProduct(array, windowSize) {
  return R.pipe(
      R.length,
      R.subtract(R.__, windowSize),
      R.range(0),
      R.reduce((accum, value) => {
          return R.pipe(
              R.slice(value, value + windowSize),
              R.product,
              v => v > accum ? v : accum
          )(array);
      }, 0)
  )(array);
}
```

Notice we did not need any loops or if statements.

Drawbacks of Ramda & FP

1. Function pipelines can be more difficult to debug
2. Performance - immutability comes at a price
3. Steeper learning curve
4. The occassional cryptic error message

Using Ramda in Production - Social Native

How and where is SN using Ramda?

All over the place! Great on both frontend or backend.

Especially useful for:

1. Data processing
2. Data tranformations

Real life example - object to CSV transforms

We are often transforming data blobs to csv to generate reports. How does it work?

For example, we want to transform:

```
{
    field1: 'boop',
    parent: {
        field2: 'bop',
        field3: 'it'
    }
}
```

to:

```
{
    displayName1: 'boop',
    displayName2: 'bop_it'
}
```

Describe the shape and transformations

```
import {getLeafPaths} from '...';

const TEMPLATE_LEAF_MARKER = 'TEMPLATE_LEAF_MARKER';
const BASE_TEMPLATE_METADATA = { TEMPLATE_LEAF_MARKER };
const DATA_NOT_AVAILABLE_PLACEHOLDER = '--';

const EXAMPLE_TRANSFORM_TEMPLATE = {
    "field1": R.merge({'displayName': 'displayName1'}, BASE_TEMPLATE_METADATA),
    "parent": R.merge(
        {
            'displayName': 'displayName2',
            'processor': (parent) => parent.field2 + '_' + parent.field3
        },
        BASE_TEMPLATE_METADATA
    )
};
```

The goods

```
function transformBlobToCSVRow(blobToTransform, transformTemplate) {
    const allLeaves = getLeafPaths(transformTemplate);
    const findPathsWithTemplateLeafMarker = R.filter(
        R.pipe(
            R.last,
            R.equals(TEMPLATE_LEAF_MARKER)
        )
    );
    const removeTemplateLeafMarkerFromPath = R.init;

    const pathsToTransform = R.map(
        removeTemplateLeafMarkerFromPath,
        findPathsWithTemplateLeafMarker(allLeaves)
    );

    const getProcessorFn = R.ifElse(
        R.has('processor'),
        R.prop('processor'),
        R.always(R.identity)
    );

    ...
```

The goods cont...

```
    ...
    const transformedBlob = R.reduce(
        (acc, transformPath) => {
            const transformMetadata = R.path(transformPath, transformTemplate);

            const displayName = R.ifElse(
                R.has('displayName'),
                R.prop('displayName'),
                R.always(R.join('_', transformPath))
            )(transformMetadata);

            const processorFn = getProcessorFn(transformMetadata);
            const valueToProcess = R.path(transformPath, blobToTransform);

            const value = R.ifElse(
                R.isNil,
                R.always(DATA_NOT_AVAILABLE_PLACEHOLDER),
                processorFn
            )(valueToProcess);

            return R.assoc(displayName, value, acc);
        },
        {},
        pathsToTransform
    );

    return transformedBlob;
}
```

Realized Benefits of FP

How is this helping?

1. More readable code
2. Encourages breaking up processing into digestable pieces
3. Flatter code, easier to reason about
4. Encourages declarative vs imperative

Questions

Contact

- samuelp@socialnative.com
- aaron@leaselock.com

Links

- Presentation: http://github.com/aaron9000/js-presentation
- Ramda: http://ramdajs.com/