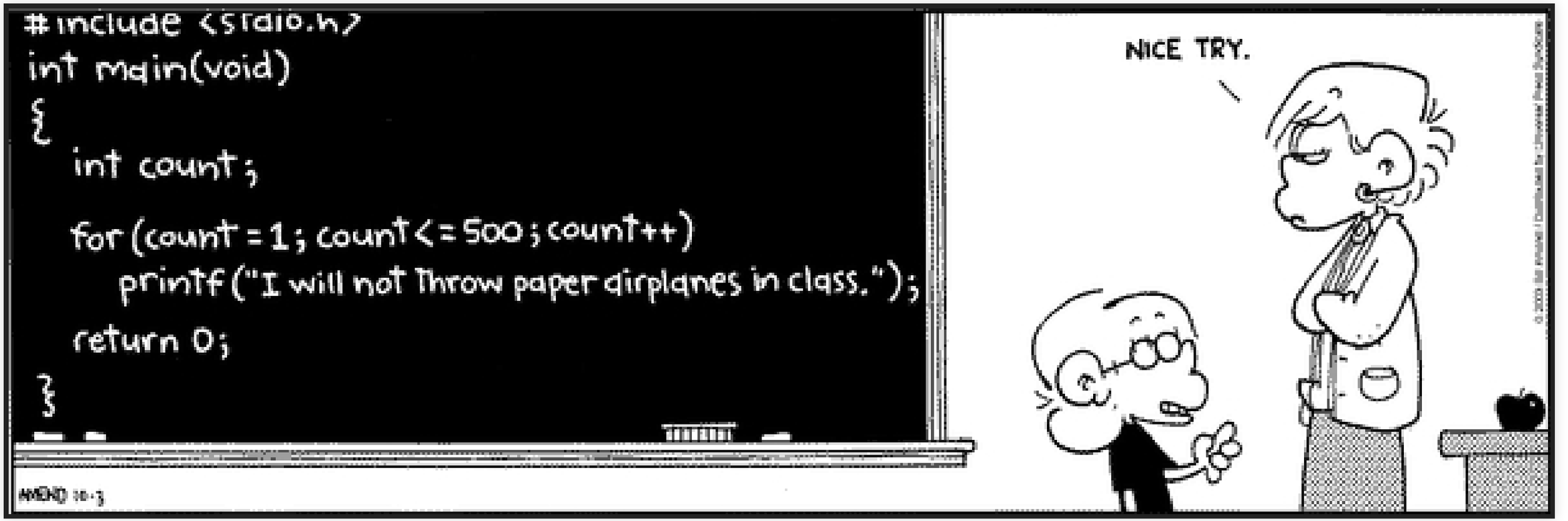
Simplified Functional JS

Elegant Resilience

2019 Connect.tech

Jeff Barczewski

- @jeffbski
- jeff@codewinds.com
- https://codewinds.com/connect-funjs


#include <stdio.h>
int main(void)
{
int count;
for (count = 1; count <= 500; count++)
printf ("I will not throw paper airplanes in class.");
return 0;
}
NICE TRY.

Jeff Barczewski

- Married, Father, Catholic
- 30 yrs (A seasoned professional :-)
- JS (since 95, primary last 8 years)
- Work: USAF, RGA, MasterCard ApplePay, Elsevier, Monsanto, Sketch
- Founded CodeWinds, live/online training, mentoring, consulting
- Languages: Fortran, C, C++, Java, Ruby, JavaScript, Go, Rust

F15 Eagle

F-15E Strike Eagle by Gerry Metzler - IMG_214 Licensed CC BY-SA 2.0

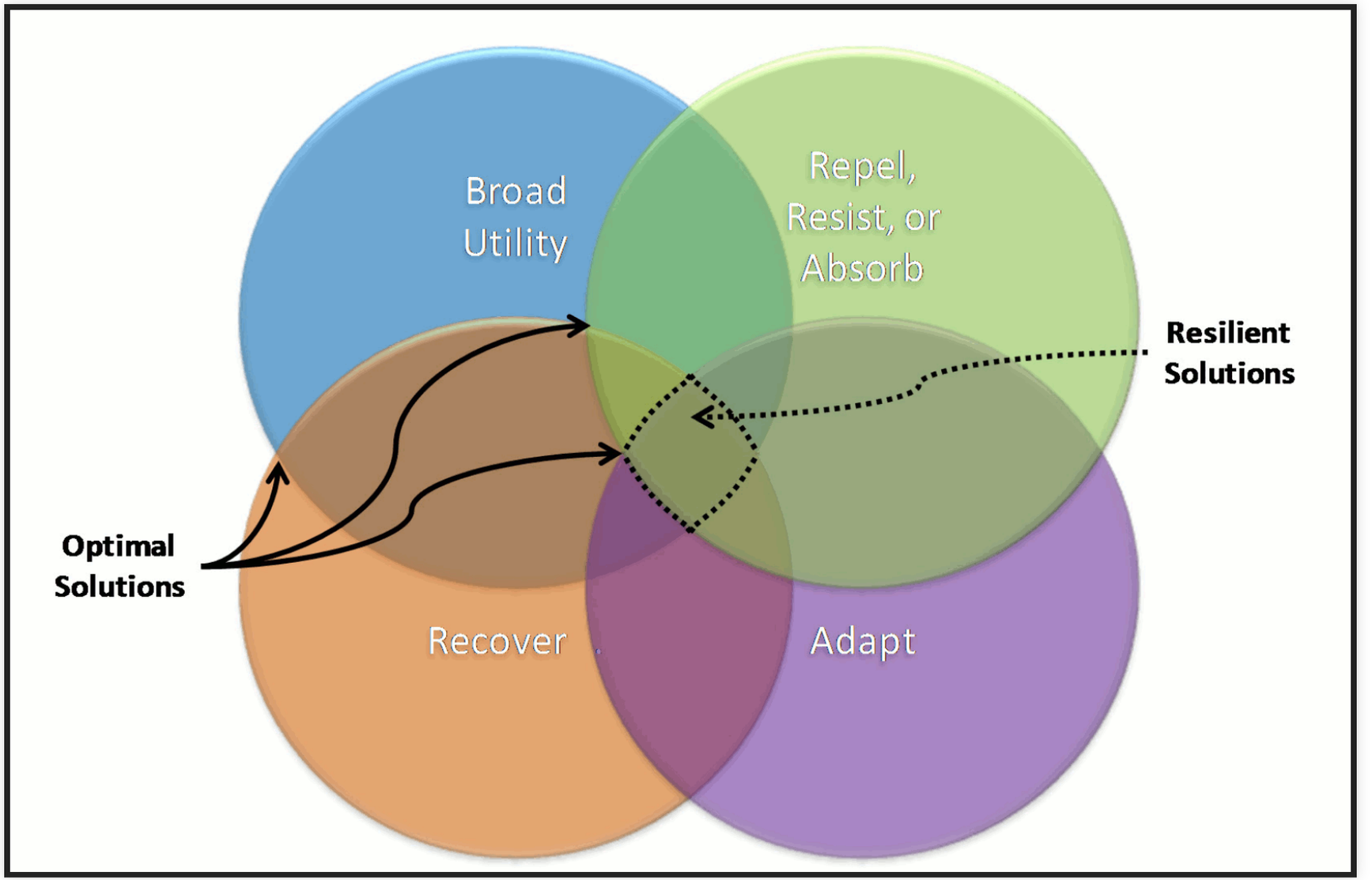
Afghanistan, F-15E 391st" by Staff Sgt. Aaron Allmon (USAF) - Src. Public domain

F15 Single Wing Landing

F-15 Single wing by Israeli Defense Force

F-15 Single wing landing by History Channel

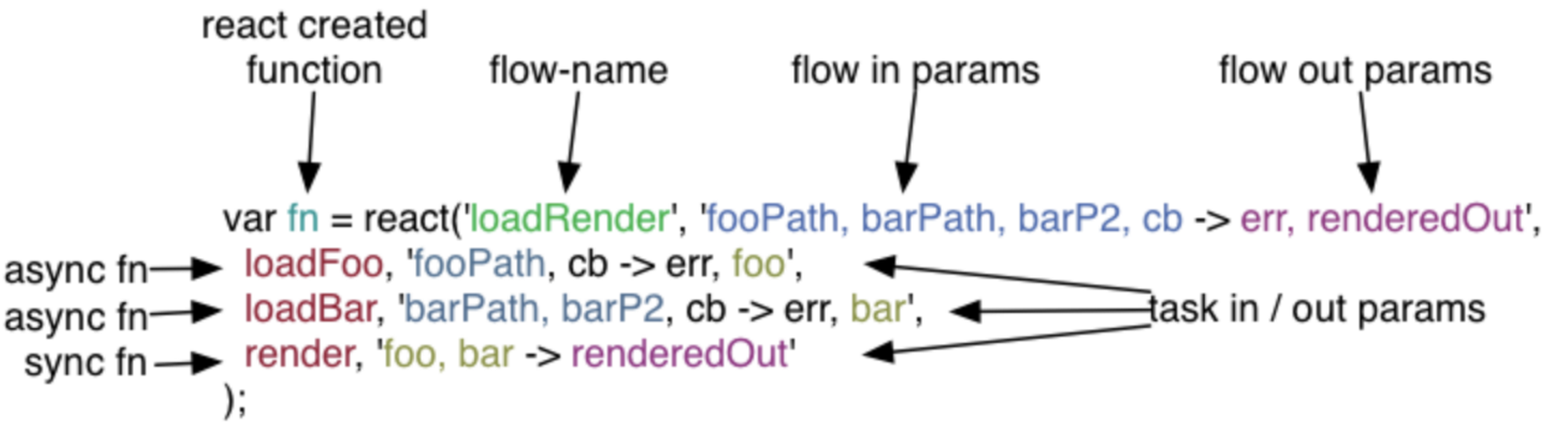
Resilience


Broad
Utility
Repel,
Resist, or
Absorb
Resilient
Solutions
Optimal
Solutions
Recover
Adapt

Exploring Code - OOP Style 2011

It takes the following arguments to define a flow function:

```
var fn = autoflow('loadRender', 'fooPath, barPath, barP2, cb -> err, renderedOut',
  loadFoo, 'fooPath, cb -> err, foo',
  loadBar, 'barPath, barP2, cb -> err, bar',
  render, 'foo, bar -> renderedOut'
);
```



Exploring Code - OOP Style 2011

```
function BaseTask() {}

BaseTask.prototype.getOutParams = function() {
  return array(this.out); // ensure array
};

BaseTask.prototype.isComplete = function() {
  return this.status === STATUS.COMPLETE;
};

BaseTask.prototype.start = function(args) {
  this.args = args;
  this.env.currentTask = this;
  this.env.flowEmitter.emit(EventManager.TYPES.EXEC_TASK_START, this);
};

BaseTask.prototype.complete = function(args) {
  this.status = STATUS.COMPLETE;
  this.results = args;
  this.env.currentTask = this;
  this.env.flowEmitter.emit(EventManager.TYPES.EXEC_TASK_COMPLETE, this);
};
```

Exploring Code - OOP Style 2011

```
CbTask.prototype = new BaseTask();
CbTask.prototype.constructor = CbTask;
CbTask.prototype.prepare = function prepare(handleTaskError, vCon, contExec) {
  var self = this;
  this.cbFun = function(err, arg0, arg1, argn) {
    var args = Array.prototype.slice.call(arguments, 1);
    if (err) {
      handleTaskError(self, err);
      return;
    }
    vCon.saveResults(self.out, args); //no error, save, continue
    self.complete(args);
    contExec();
  };
};

CbTask.prototype.exec = function exec(vCon, handleError, contExec) {
  try {
    var args = this.a.map(function(k) {
      return vCon.getVar(k);
    }); //get args from vCon
    this.start(args); //note the start time, args
    args.push(this.cbFun); // push callback fn on end
```

What could we do different?

- Not use inheritance
- Define all the state in a constructor
- Maybe there is another way altogether?

Could we use functions?

- Enacpsulate operations in functions
- Break complex functions into smaller ones
- Each function doing one thing

JS Functions (2019)

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

const bar = (d, e) => d - e;
const cat = (f, g) => {
  return f * g;
};

// functions are first class citizens
// can be passed around to functions and assigned to variables
const alsoFoo = foo;
const add = (j, k) => j + k;
const subtract = (j, k) => j - k;

function exec(operation, a, b) {
  return operation(a, b);
}
const result = exec(add, 10, 2); // 12
const result2 = exec(subtrace, 10, 2); // 8
```

JS Functions (2019) - Closures

```
// todo.js
const todos = [];

export function addTodo(todo) {
  todos.push(todo);
}

export function displayTodos() {
  console.log(todos);
}
```

```
import { addTodo, displayTodos } from "./todo";

addTodo("buy milk");
addTodo("buy eggs");
addTodo("buy bread");
displayTodos();
// ['buy milk', 'buy eggs', 'buy bread']
```

What if we try to just use only the inputs?

```
// todo.js
export function addTodo(todos, todo) {
  todos.push(todo);
  return todos;
}
export function displayTodos(todos) {
  console.log(todos);
}
```

```
import { addTodo, displayTodos } from "./todo";
let todos = [];
todos = addTodo(todos, "buy milk");
todos = addTodo(todos, "buy eggs");
todos = addTodo(todos, "buy bread");
displayTodos(todos); // ['buy milk', 'buy eggs', 'buy bread']
todos = addTodo(todos, "make french toast");
displayTodos(todos);
// ['buy milk', 'buy eggs', 'buy bread', 'make french toast']
```

Find the bug

```
// todo.js
export function addTodo(todos, todo) {
  todos.push(todo);
  return todos;
}
export function displayTodos(todos) {
  while (let todo = todos.shift()) {
    console.log(`- ${todo}`);
  }
}
```

```
import { addTodo, displayTodos } from "./todo";
let todos = [];
todos = addTodo(todos, "buy milk");
todos = addTodo(todos, "buy eggs");
todos = addTodo(todos, "buy bread");
displayTodos(todos); // ['buy milk', 'buy eggs', 'buy bread']
todos = addTodo(todos, "make french toast");
displayTodos(todos); // ['make french toast']
```

Pure functions

```
// todo.js
export function addTodo(todos, todo) {
  return todos.concat(todo); // return a new array
}
export function displayTodos(todos) {
  // not modifying array
  todos.forEach(todo => {
    console.log(`- ${todo}`);
  });
}
```

```
import { addTodo, displayTodos } from "./todo";
let todos = [];
todos = addTodo(todos, "buy milk");
todos = addTodo(todos, "buy eggs");
todos = addTodo(todos, "buy bread");
displayTodos(todos);
// ['buy milk', 'buy eggs', 'buy bread']
todos = addTodo(todos, "make french toast");
displayTodos(todos);
// ['buy milk', 'buy eggs', 'buy bread', 'make french toast']
```

Dealing with side effects

```
function writeToStorage(data) {
  // write to file, db, ...
}

const add = (a, b) => a + b;
const subtract = (a, b) => a - b;

function app(writeToStorage, a, b) {
  const sum = add(a, b);
  writeToStorage(sum);
}
app(writeToStorage, 10, 2); // writes 12 to storage

// can use destructuring if many args to pass in
function app({ writeToStorage, operation }, a, b) {
  const sum = operation(a, b);
  writeToStorage(sum);
}
app({ writeToStorage, operation: add }, 10, 2); // writes 12
app({ writeToStorage, operation: subtract }, 10, 2); // writes 8
```

Functional Programming Guidelines

- Use pure functions as much as you can
  - Only operate on the inputs and return new outputs
  - They always return same result if passed same input
  - Don't mutate data
- When you have to make side effects, isolate them to functions and pass them near the top.

Serverless / FaaS

- Deploying functions directly
- Auto scale independently
- No OS or container to worry about

When to use FP style?

- Anywhere
- My preferred style
- Can work along side OOP, introduce over time
- Replace complex, troublesome code with functions
- Break into smaller functions

How? - working with data in immutable fashion

```
// pure JS
// arrays
const arr = [1, 2];
const arr2 = arr.concat(3); // [1, 2, 3]
const arr3 = [...arr, 3]; // [1, 2, 3];

// objects
const obj = { a: 1, b: 2 };
// object spread into new object
const obj2 = {
  ...obj, // spreads a and b
  c: 3
};
// { a: 1, b: 2, c: 3}
```

ES5 features - map, filter, reduce, forEach

```
const arr = [10, 20, 30, 40];
const result = arr
  .filter(x => x > 20) // [30, 40]
  .map(x => x + 1) // [31, 41]
  .reduce((acc, x) => {
    return acc + x;
  }, 0);
// 72
```

Tools to help

- eslint - use rule sets to help you avoid bugs
  - instantly get feedback in your editor
  - also use during CI builds to verify quality of the code
  - standardJS is a nice ruleset to get started with
- eslint-plugin-fp - allows you to warn or error if you mutate data
  - eslint-config-hardcore-fp
  - eslint-plugin-js-fp-linter
  - eslint-config-standard-fp
- prettier - auto format your code consistently
  - eslint-config-prettier-standard - to use standardjs and prettier combined

ImmutableJS

- https://github.com/immutable-js/immutable-js
- Created by Lee Byron, Facebook
- Implements Persistent Data Structures in JS
  - Structural sharing
- Intuitive methods for manipulating data
- Returns new object after each operation, original is untouched

lodash/fp

Functional Programming style interface to lodash

- immutable - creates new data, old untouched
- auto-curried - returns partially applied fn's
- data last argument order - for easier composition
- https://lodash.com/ https://github.com/lodash/lodash/wiki/FP-Guide
- Generated FP docs - https://gist.github.com/jfmengels/6b973b69c491375117dc

```
# Installation:
npm install lodash
```

```
// Usage:
import { set, update } from "lodash/fp";
```

lodash/fp set

```
import { set } from "lodash/fp";

// previous state is unchanged after each set
const state0 = { a: 1 };
const state1 = set("b.bb", "hello", state0); // { a: 1, b: { bb: 'hello' }}
const state2 = set(["c", "cc"], 123, state0); // { a: 1, c: { cc: 123 }}
const state3 = set("d.dd[0]", "hey", state0); // { a: 1, d: { dd: ['hey'] }}
const state4 = set("a", 234, state3); // { a: 234, d: { dd: ['hey'] }}
const state5 = set("e.ee.eee", 789, null); // {e: { ee: {eee: 789}}}
```

lodash/fp update

```
import { update } from "lodash/fp";

// previous state is unchanged after each set
const state0 = { a: { count: 1, greet: "hello" }, b: [true] };
const state1 = update("a.count", x => x + 1, state0);
// { a: {count: 2, greet: 'hello'}, b: [true]}

const state2 = update("b", arr => arr.concat(false), state1);
// { a: {count: 2, greet: 'hello'}, b: [true, false]}

const state3 = update(["a", "greet"], x => x + " world", state2);
// { a: {count: 2, greet: 'hello world'}, b: [true, false]}
```

lodash/fp get, getOr

```
import { get, getOr } from "lodash/fp";

const state0 = { a: 1, b: { bb: { bbb: 123 } } };
const bbb = get("b.bb.bbb", state0); // 123
const fnGetBBB = get("b.bb.bbb");
const bbb2 = fnGetBBB(state0); // 123
const cc = get("c.cc", state0); // undefined

const cc2 = getOr("mydefault", "c.cc", state0); // 'mydefault'
const a = getOr("hey", "a", state0); // 1
const fnGetA = getOr("hey", "a");
const a2 = fnGetA(state0); // 1
const a3 = fnGetA({}); // 'hey'
const a3 = fnGetA(null); // 'hey'
```

lodash/fp pick, omit

```
import { pick, omit } from "lodash/fp";

const state0 = { a: 1, b: { bb: { bbb: 123 } }, c: true };
const state1 = pick("a", state0); // {a: 1}
const state2 = pick(["a", "c"], state0); // {a: 1, c: true}

const state3 = { b: { bb: { bbb1: 123, bbb2: "hello" } } };
const state4 = pick("b.bb.bbb1", state3); // {b: {bb: {bbb1: 123}}}

const state5 = omit("a", state0); // { b: { bb: { bbb: 123 } }, c: true };
const state6 = omit(["a", "c"], state0); // { b: { bb: { bbb: 123 } } };
```

lodash/fp partial, curry

```
import { curry, partial } from "lodash/fp";

function add3(a, b, c) {
  return a + b + c;
}
const fn = partial(add3, [10, 20]);
const result = fn(30); // 60

const add3Curry = curry(add3);
const fn2 = add3Curry(10);
const fn3 = fn2(20);
const result2 = fn3(30); // 60

const fn4 = add3Curry(10, 20);
const result3 = fn4(30); // 60
```

lodash/fp compose, flow

```
import { compose, flow } from "lodash/fp";

const incr = x => x + 1;
const multiplyBy10 = x => x * 10;

const a = incr(multiplyBy10(20)); // 201
// right to left
const fn1 = compose(
  incr,
  multiplyBy10
);
const a2 = fn1(20); // 201

// left to right
const fn2 = flow(
  multiplyBy10,
  incr
);
const a3 = fn2(20); // 201
```

lodash/fp flow, partial, update

```
import { flow, partial, update } from "lodash/fp";

const state0 = { sum: 100, count: 2 };
const add = (a, b) => a + b;
function addItem(item, state) {
  const addVal = partial(add, [item]);
  return flow(
    update("sum", incrByVal),
    update("count", x => x + 1)
  )(state);
}
const state1 = addItem(23, state0); // {sum: 123, count: 3}
```

Ramda

- Some overlap with lodash/fp
- Has additional functionality for working with FP
- Not yet tuned for performance
- https://ramdajs.com/

React, Redux, Recompose

- React - UI library supporting a functional style
  - Stateless function components, just functions
  - Pass in data, it figures out optimal operations to update DOM
- Redux - library for dealing with state in functional way
- Recompose - libary for composing together React components

```
function Orders({ orders }) {
  return (
    <ul>
      {orders.map(x => (
        <li key={x.id}>
          {x.name} - {x.date}
        </li>
      ))}
    </ul>
  );
}
```

RxJS

- https://github.com/ReactiveX/rxjs
- Deal with data/events over time
- Similar feel to map, filter, reduce
- Create a pipeline of operations

Other FP languages that compile to JS

- Clojurescript - Clojure lisp syntax
- Elm
- Purescript - Haskell like
- Reason - OCaml (Facebook)
- Clojurescript, Purescript, Reason will work nicely with React. Elm has React/Redux built-in. Elm doesn't let you access JS libs directly need to create a port.

Summary

- Use functions
- Strive for pure functions which don't mutate data and produce side effects
- Compose simple functions together to handle complex problems
- Isolate side effects into functions and pass those in near the top
- Treat data immutably, returning new objects, arrays
- Use eslint, eslint-plugin-fp to catch mutations
- Use lodash/fp and other tools to simplify FP style programming

Continued Learning about FP

- Simple made Easy - talk by Rich Hickey at Strange Loop
  - https://www.infoq.com/presentations/Simple-Made-Easy
- Out of the Tar Pit - Reducing complexity in large-scale software systems
  - https://github.com/papers-we-love/papers-we-love/blob/master/design/out-of-the-tar-pit.pdf
- Strange Loop conference and videos
  - https://www.thestrangeloop.com/
  - https://www.youtube.com/channel/UC_QIfHvN9auy2CoOdSfMWDw

Thanks

- https://codewinds.com/connect-funjs (slides, resources)
- https://codewinds.com/ (blog, tips/training, consulting)
- jeff@codewinds.com
- @jeffbski