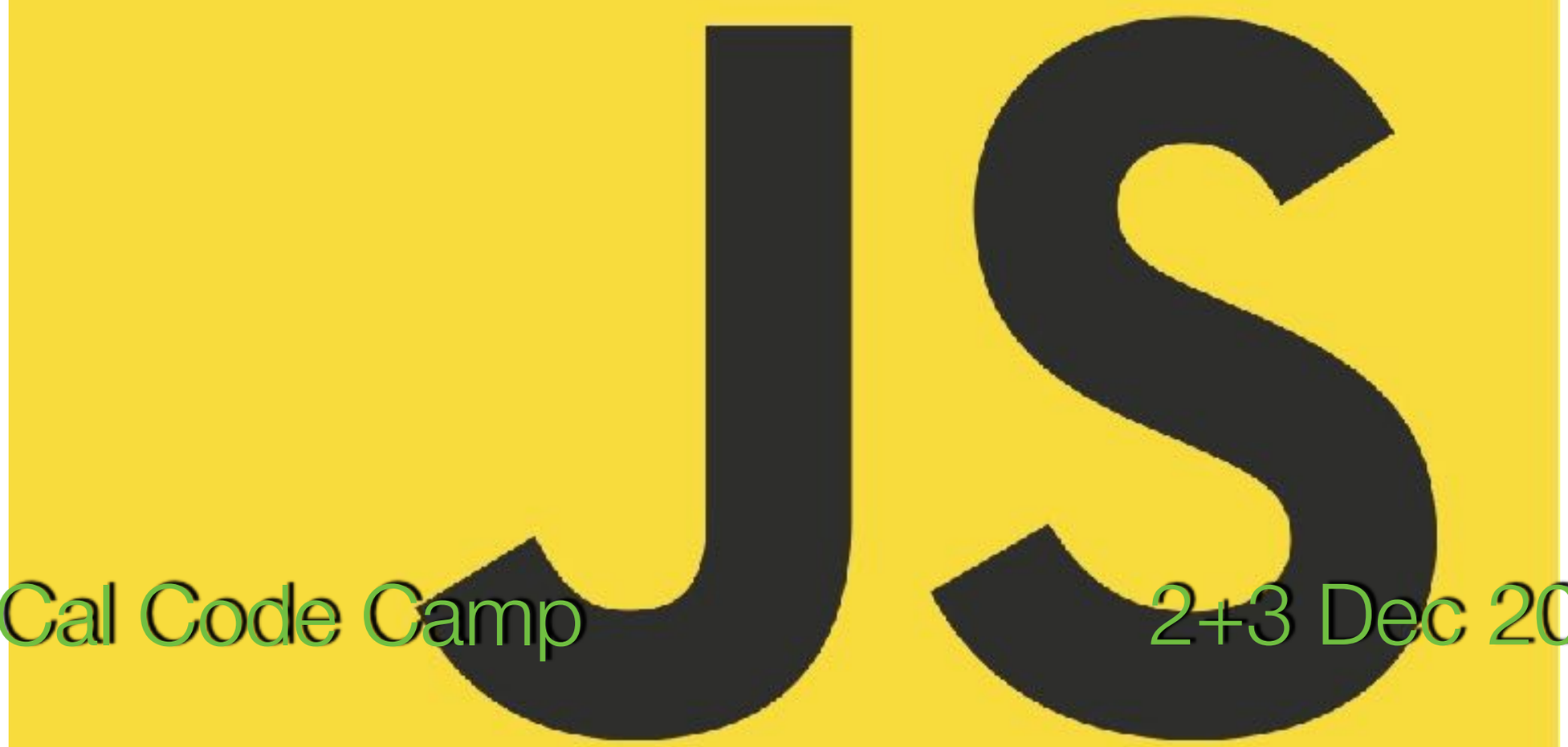
FizzBuzz in Functional JavaScript



SoCal Code Camp 2+3 Dec 2017

Troy Miles

- Troy Miles
- Nearly 40 years of experience
- Programmer, speaker & author
- rockncoder@gmail.com
- @therockncoder
- lynda.com Author

Code + Slides Online

- https://github.com/Rockncoder/FP-JS

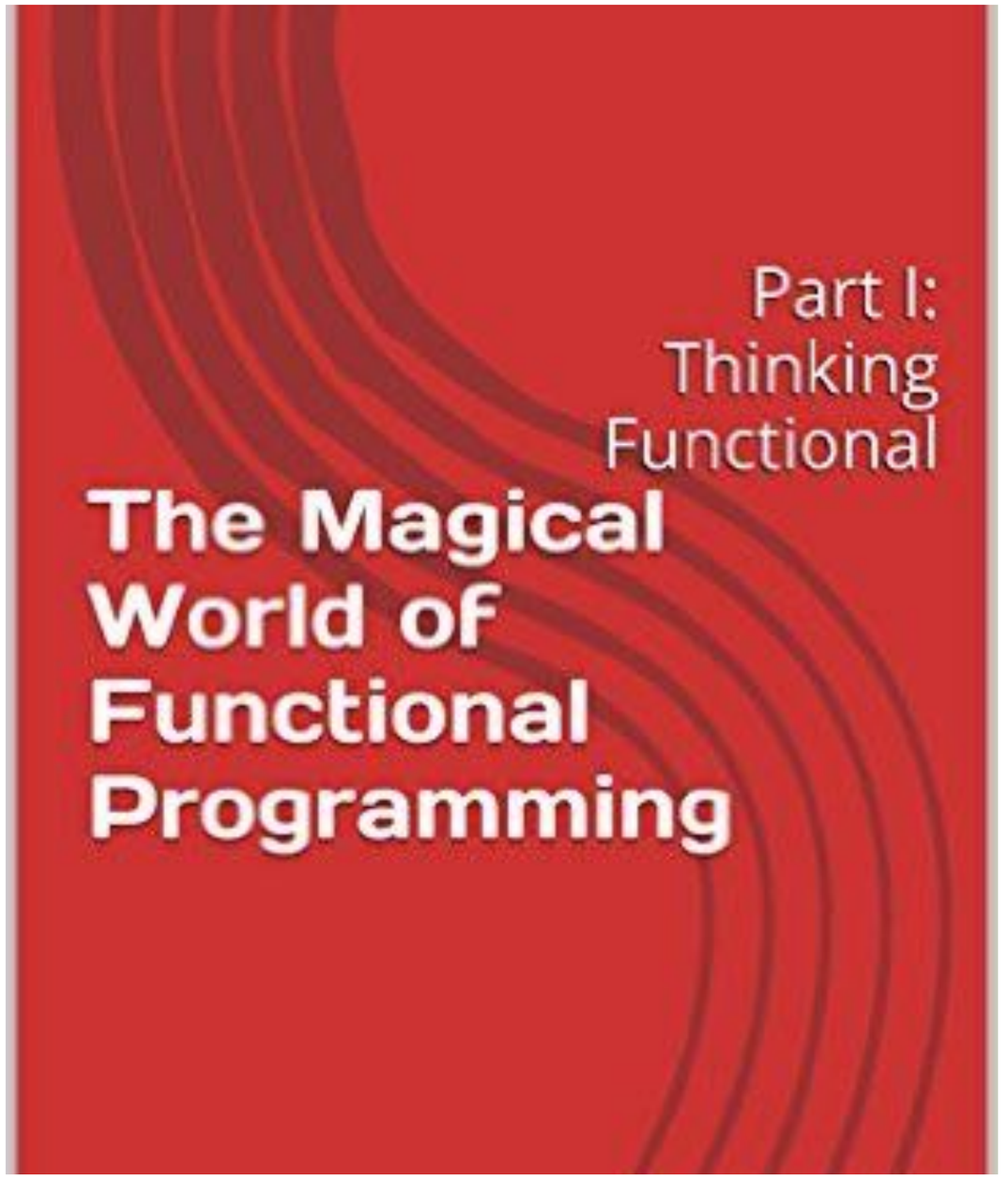
- http://www.slideshare.net/rockncoder

The book

- The Magical World of Functional Programming: Part 1: Thinking Functionally

- by K Anand Kumar

- http://amzn.to/1HPzRro



FizzBuzz
Requirements

- Print the numbers 1 to 100
- On multiples of 3 print “Fizz” instead of the number
- On multiples of 5 print “Buzz” instead of the number
- For multiples of 3 and 5 print “FizzBuzz”

fizz buzz - chapter 1

Just make it work

- The code looks like it came from a 1st year computer-sci majors homework assignment

- It works, but it you really have to think about what it does and why

```
12      // this is not a function, it is a procedure
13      const fizzBuzz = function () {
14        for (let i = 1; i <= 100; i += 1) {
15          let printVal = i + ' ';
16          if (i % 3 === 0) {
17            printVal += 'Fizz';
18          }
19          if (i % 5 === 0) {
20            printVal += 'Buzz';
21          }
22          console.info(printVal);
23        }
24      };
25
26      fizzBuzz();
```

What is Functional Programming?

Key Functional Features

- Pure functions
- First-class / High order functions
- Immutable data
- Recursion
- Referential transparency

Functional vs. Imperative

what?	functional	imperative
primary construct	function	class instance
state change	bad	important
order of execution	not important	important
flow control	function calls recursion	loops, conditionals, method calls

Sample Languages

mostly functional	**mixed**	**mostly imperative**
Lisp/Scheme	JavaScript	Java
ML	Scala	C#
Haskell	Python	C++
Clojure	Dart	Swift
F#	Lua	Ruby
Erlang	R	Kotlin

Pure Functions

- Must return a value
- Can’t produce any side-effects
- Must return the same output for a given input

Pure Functions Are Super

- Cacheable
- Portable
- Self-documenting
- Testable
- Reasonable

Pure Function

```
5     const minimum = 21;
6     // not pure since it relies on value outside of it
7     function impure(age) {
8       return age >= minimum;
9     }
10
11    // pure since it is self contained
12    function pure(age) {
13      const minimum = 21;
14      return age >= minimum;
15    }
```

First-Class Functions

- Treats functions as first class citizens
- Assigned to variables & stored in arrays
- Anonymous and nested functions
- Higher-order functions

Higher-Order Functions

- Takes one or more functions as arguments
- Or returns a function as its results
- Allows for the creation of function factories
- This is the core of the curry function

fizz buzz - chapter 2
Separation of concerns

- It could be argued that we made things worse
- Our 12 line program is now 36 lines (with comments and spacing)
- But it is easier to understand

```
  // 1: a controller since it is in charge
  // let's take the range as parameters instead of hard coding
  const controller = function (from, to) {
    for (let i = from; i < to; i++) {
      print(formatOutput(test(i)));
    }
  };
  // 2: a tester
  const test = function (num) {
    let retval = [num];
    if (num % 3 === 0) {
      retval.push('Fizz');
    }
    if (num % 5 === 0) {
      retval.push('Buzz');
    }
    return retval;
  };
  // 3: formatter
  const formatOutput = function (ar) {
    let output = ar[0] + ' ';
    for (let i = 1; i < ar.length; i++) {
      output += ar[i];
    }
    return output;
  };
  // 4: display the results
  const print = function (output) {
    console.info(output);
  };

  controller(1, 100);
```

Write your code to be read by humans.

ES5 Array Methods

- .isArray()
- .every()
- .forEach()
- .indexOf()
- .lastIndexOf()
- .some()
- .map()
- .reduce()
- .filter()

map

```
const letters = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
// MAP
// iterates over the elements and returns an equal sized new array
const times2 = numbers.map(element => element * 2);
console.log(times2);
// [ 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

filter

```
2   const letters = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];
3   const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
9   // FILTER
10  // iterates over the elements and a creates new array with those that pass the test
11  const evenOnly = numbers.filter(element => !(element % 2));
12  console.log(evenOnly);
13  // [ 2, 4, 6, 8, 10]
```

reduce

```
2    const letters = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];
3    const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
14   // REDUCE
15   // turns a collection into a scalar value
16   const letters2 = letters.reduce((previous, current) => previous + current);
17   console.log(letters2);
18   // ABCDEFG
19   let letters3 = letters.reduceRight((previous, current) => previous + current);
20   console.log(letters3);
21   // GFEDCBA
```

forEach

```
6       const nums = [2, 4, 6, 8, 10];
7       // forEach iterates over the array, but there is no way to break out
8       nums.forEach(function (elem, index) {
9         console.log(index + ': ' + elem);
10      });
11      //   0: 2
12      //   1: 4
13      //   2: 6
14      //   3: 8
15      //   4: 10
```

fizz buzz - chapter 3
Hidden jewels from ES5

- In this version we take advantage of the array methods from ES5

- The controller uses the map and forEach methods

- We use the reduce method to de-clutter formatOutput()

side bar

A little more of array.from

- Creates arrays from something else
- Can convert a string, arguments, or anything with a length and indexed elements
- We use it to create our initial array

Array.from
ES6 Data factory

- Creates a new array from an array-like or iterable object
- Makes arrays from strings, sets, maps, and ***arguments,***
- And from a passed function

```
const instantArr = Array.from({length: 20}, (_, index) => index + 1);
console.info(instantArr);

// we can create an array from a string
const codersArray = Array.from("happy coders");
console.log(codersArray);

// .entries() returns the index and the element of each array item
for (let [index, elem] of codersArray.entries()) {
  if (index === 5) {
    break;
  }
  console.log(`${index}. ${elem}`);
}
console.log(``);

// unlike forEach, continue and break work
for (let [index, elem] of codersArray.entries()) {
  if (index === 5) {
    continue;
  }
  console.log(`${index}. ${elem}`);
}
```

```
    // replacing loop with a map, passing in the range
    const controller = function (from, to) {
      // we generate the array using ES2015
      const ar = Array.from({length: to - from + 1}, (elem, index) => index +
1);
      // we map over the array to generate the results array
      const results = ar.map((elem, index) => formatOutput(test(elem)));
      // and finally we render the results
      results.forEach(print);
    };
    // unchanged from the last iteration
    const test = function (num) {
      let retval = [num];
      if (num % 3 === 0) {
        retval.push('Fizz');
      }
      if (num % 5 === 0) {
        retval.push('Buzz');
      }
      return retval;
    };
    // this is a reduction - taking an array and making it a single string
    const formatOutput = function (ar) {
      return ar.reduce((prev, curr, ndx) => ndx === 1 ? prev + ' ' + curr :
prev + curr, '');
    };
    // unchanged from the last iteration
    const print = function (output) {
      console.info(output);
    };

    controller(1, 100);
```

fizz buzz - chapter 4
Harnessing functions

- The test function has been a bit of a mess
- We break into 3 functions, initialize, fizz, and buzz
- Each function is descriptive and we follow the flow

```
15      // 1: convert the number into an array
16      const initialize = function (x) {
17        return [x];
18      };
19      // 2: appends Fizz if the number is a multiple of 3
20      const fizz = function (tuple) {
21        if (tuple[0] % 3 === 0) {
22          tuple.push('Fizz');
23        }
24        return tuple;
25      };
26      // 3: appends Buzz if the number is a multiple of 5
27      const buzz = function (tuple) {
28        if (tuple[0] % 5 === 0) {
29          tuple.push('Buzz');
30        }
31        return tuple;
32      };
33      // 4: use composition to combine it all together
34      const test = function (num) {
35        return buzz(fizz(initialize(num)));
36      };
```

fizz buzz - chapter 5
Fun, fun, function factory

- The code from our last chapter had redundancies
- Each test was the same except, test value and the message
- A function factory now creates the test for us
- The technical term is a *partial application*

```
15    // function factory to test conditions
16    const testMaker = function (condition, whenTrue) {
17      return function (tuple) {
18        if (condition(tuple[0])) {
19          tuple.push(whenTrue);
20        }
21        return tuple;
22      }
23    };
24    // using es2015 arrow functions since they are prettier
25    const initialize = x => [x];
26    const fizz = testMaker(x => x % 3 === 0, 'Fizz');
27    const buzz = testMaker(x => x % 5 === 0, 'Buzz');
28    const test = num => buzz(fizz(initialize(num)));
```

function factories
The magic of closures

- Another example of a function factory
- This one also use a closure
- The value of ’n’ is closed over

```
// a simple function factory
function addN(n) {
  // a function is returned
  return function (x) {
    return n + x;
  }
}

// let's create two functions, add10 and add50
const add10 = addN(10);
const add50 = addN(50);
// they don't interfere with each other or share variables or state
console.info(add10(1));
console.info(add50(1));
console.info(add10(5));
// 11
// 51
// 15
```

fizz buzz - chapter 6

Move to a higher level

- Functional programming is a level of abstraction above imperative
- We focus on what we want, not so much how to it
- Now we easily add a new test

```
const initialize = x ⇒ [x];
const fizz = testMaker(x ⇒ x % 3 === 0, 'Fizz');
const buzz = testMaker(x ⇒ x % 5 === 0, 'Buzz');
// adding a bang function is easy
const bang = testMaker(x ⇒ x % 7 === 0, 'Bang');
const test = num ⇒ bang(buzz(fizz(initialize(num))));

const formatOutput = function (ar) {
  return ar.reduce((prev, curr, ndx) ⇒ ndx === 1 ? prev + ' ' + curr : prev +
curr, '');
};
```

fizz buzz - chapter 7
Better together

- Another functional concept is composition
- It allows us to combine functions together to build new functions
- We use it to combine our test and format functions

```
    // here is are simple compose function
    const compose = function (fn1, fn2) {
      return function (arg) {
        return fn2(fn1(arg));
      }
    };
    // let's use compose to combine the test and format functions
    const controller = function (from, to, testFunc, formatFunc, outputFunc)
{
      const ar = Array.from({length: to - from + 1}, (elem, index) => index +
1);
      const mapFunc = compose(testFunc, formatFunc);
      const results = ar.map((elem, index) => mapFunc(elem));
      results.forEach(outputFunc);
    };

    const initialize = x => [x];
    const fizz = testMaker(x => x % 3 === 0, 'Fizz');
    const buzz = testMaker(x => x % 5 === 0, 'Buzz');
    const bang = testMaker(x => x % 7 === 0, 'Bang');
    const test = num => bang(buzz(fizz(initialize(num))));
```

Prefer composition over inheritance.

fizz buzz - chapter 8

Are we done yet?

- We have a created another partial application to create our controller
- We pass it the test, format, and print functions
- We should probably stop here

```
    // this function creates a controller
    const controllerMaker = function (testFunc, formatFunc, outputFunc) {
      return function (from, to) {
        const ar = Array.from({length: to - from + 1}, (elem, index) => index
+ 1);
        const mapFunc = compose(testFunc, formatFunc);
        const results = ar.map((elem, index) => mapFunc(elem));
        results.forEach(outputFunc);
      };
    };
    // unchanged from the last iteration
    const initialize = x => [x];
    const fizz = testMaker(x => x % 3 === 0, 'Fizz');
    const buzz = testMaker(x => x % 5 === 0, 'Buzz');
    const bang = testMaker(x => x % 7 === 0, 'Bang');
    const boop = testMaker(x => x % 11 === 0, 'Boop');
    const test = num => boop(bang(buzz(fizz(initialize(num)))));
    // unchanged from the last iteration
    const formatOutput = function (ar) {
      return ar.reduce((prev, curr, ndx) => ndx === 1 ? prev + ' ' + curr :
prev + curr, '');
    };
    // unchanged from the last iteration
    const print = function (output) {
      console.info(output);
    };
    // now we even use a factory function to create our controller
    const fizzBuzz = controllerMaker(test, formatOutput, print);
    fizzBuzz(1, 106);
```

fizz buzz - chapter 9

The step too far

- We should have stopped but we didn't
- There are a lot unneeded intermediate variables
- We've chucked them and simply pass the controllerMaker() everything
- Since it returns us a function, we don't have to save it, we can just invoke it

```
    // this code is much harder to read than the previous version
    controllerMaker(compose(compose(
      x => [x], testMaker(x=>x % 3 === 0, 'Fizz')),
      testMaker(x=>x % 5 === 0, 'Buzz')),
      formatOutput, print)
    (1, 106);
```

Functional Libraries

- Partial applications, compose, and other functional constructs should be part of the language
- For this reason JavaScript isn't consider a true functional language
- You don't have code it all yourself
- There are several libraries out there to help you

Functional JS Libraries

- http://underscorejs.org/
- https://lodash.com/
- http://ramdajs.com/
- http://eliperelman.com/fn.js/

Resources

- http://amzn.to/1HPzRro
- http://sarabander.github.io/sicp/
- http://c2.com/cgi/wiki?FizzBuzzTest
- https://kangax.github.io/compat-table/es6/
- https://github.com/Rockncoder/FP-JS
- http://www.slideshare.net/rockncoder/functional-programming-in-javascript-55110091

Next Steps…

- Play with the code
- Read the “Magical World of Functional Programming”
- Read Structure and Interpretation of Computer Programs (SICP) free online

Summary

- Functional techniques can give us a higher degree of reusability of code than is normally possible with OOP
- JavaScript supports many FP concepts
- Most of those it lacks can be created

“Object-oriented programming is an exceptionally bad idea which could only have originated in California.”

–Edsger Dijkstra