Functional Javascript

Higher Order Functions, Closures, Currying, Partial Application, function composition

Higher Order Functions

- Either:
  - Take one or more functions as an input
  - Output a function
- Common Higher order functions
- General callbacks
- Returning functions
- Re-writing functions

forEach

```
// array.forEach(callback[, thisArg])
// callback
//      Function to execute for each element
//  thisArg
//      Object to use as this when executing callback
    var data = [1, 2, 3, 4];

    data.forEach(function(element, index, array) {
        console.log("a[" + index + "] = " + element);
    });

// logs:
// a[0] = 1
// a[1] = 2
// a[2] = 3
// a[3] = 4
```

every

```
// array.every(callback[, thisArg])
// callback
//      Function to test for each element
//  thisArg
//      Object to use as this when executing callback
// return Boolean
//      True if every element passes test, otherwise false
    var data = [1, 2, 3, 4];

    var result = data.every(function(element, index, array) {
        return element <= 3;
    });

// result ===  false
```

some

```
// array.some(callback[, thisArg])
// callback
//      Function to test for each element
//  thisArg
//      Object to use as this when executing callback
// return Boolean
//      True if any element passes test, otherwise false
    var data = [1, 2, 3, 4];

    var result = data.some(function(element, index, array) {
        console.log("a[" + index + "] =" + element);
        return element > 3;
    });

// result ===  true
```

filter

```
// array.filter(callback[, thisArg])
// callback
//      Function to test each element of the array
//  thisArg
//      Object to use as this when executing callback
// return Array
//      A new Array with only the elements in the array that
//     pass the call back
    var data = [1, 2, 3, 4];

    var result = data.filter(function(element, index, array) {
        return (element < 3);
    });

// result = [1, 2]
```

map

```
// array.map(callback[, thisArg])
// callback
//      Function that produces an element of the new Array from //     an element of the current one
//  thisArg
//      Object to use as this when executing callback
// return Array
//      A new Array where each element i is the result of
//       callback(originalArray[i])
    var data = [1, 2, 3, 4];

    var result = data.map(function(element, index, array) {
         // square
        return element * element;
    });

// result = [1, 4, 9, 16]
```

reduce

```
// array.reduce(callback[, initialValue])
// callback
//      Function to execute on each value in the array
//  initialValue
//      Object to be used as the first argument to the first call of the call back
// return Anything
//      A new Array with only the elements in the array that pass the call back
    var data = [1, 2, 3, 4];

    var result = data.reduce(function(previousValue, currentValue, index, array) {
        return previousValue + currentValue;
    });

// result = 10;

    var result2 = data.reduce(function(previousValue, currentValue, index, array) {
        return previousValue + currentValue;
    }, 10);

    // result 20;
```

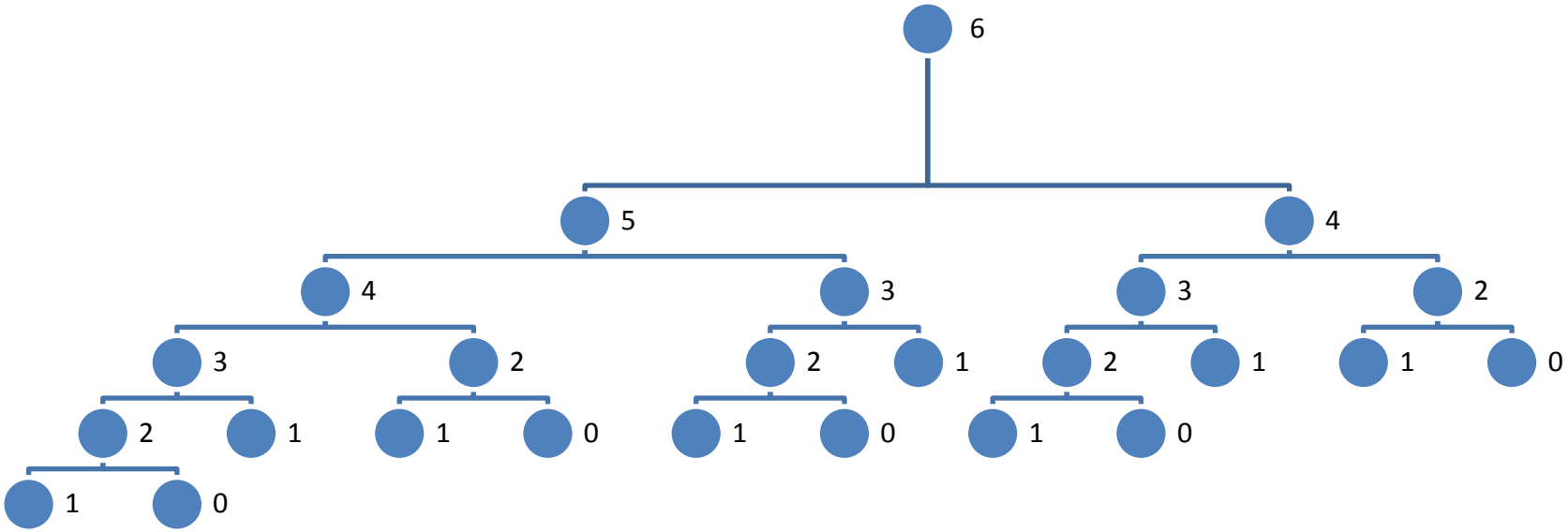
Returning a Function and closures

```
// Example of closures and returning a function
    var counter = (function() {
        var count = 0;

        return function() {
            console.log(++count);
        };

    })();

    // counter(); count = 1
    // counter(); count = 2
    // ...
    // counter(); count = n
    // count; => ReferenceError: count is not defined
    // count is in closure scope
```

More concrete example

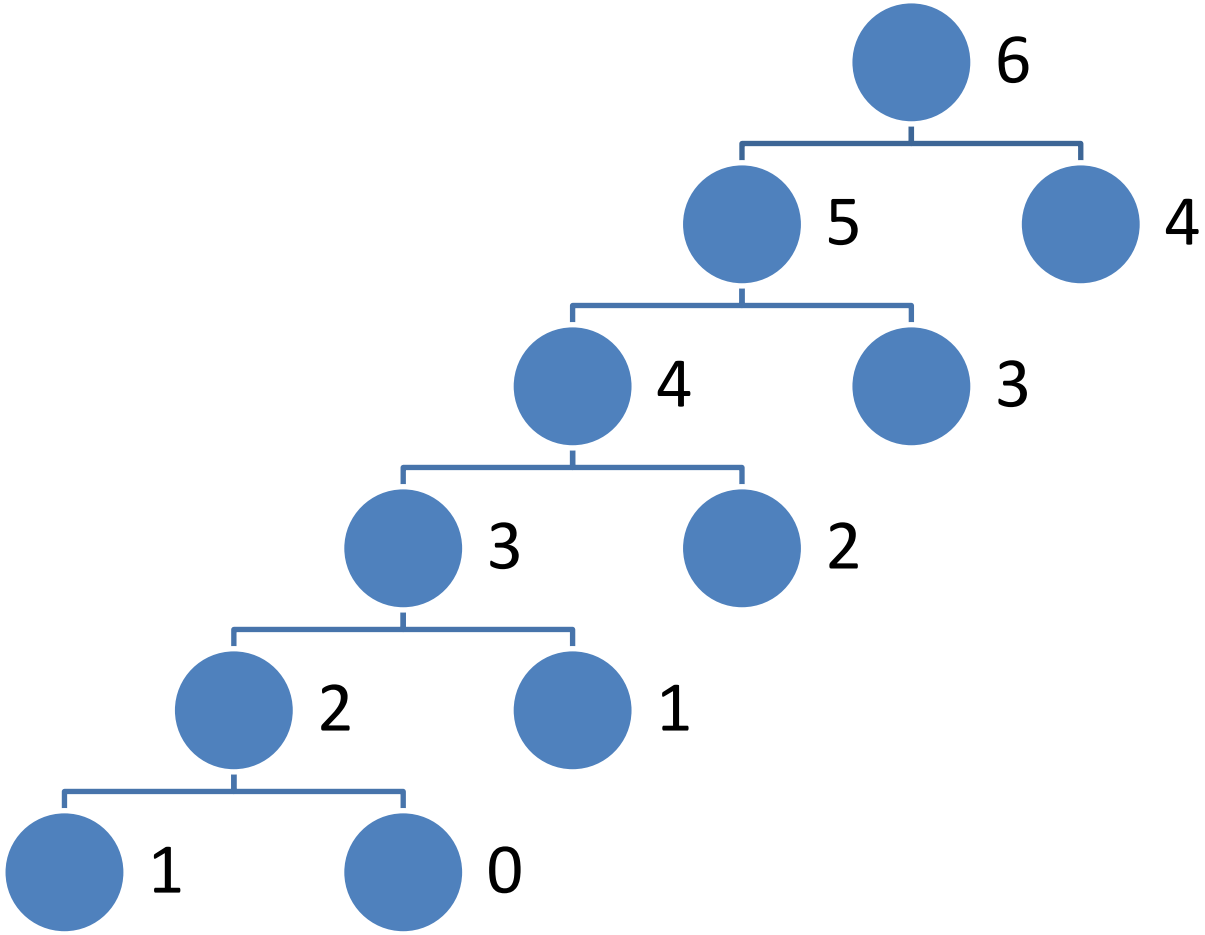
```
// standard fibonacci
function slowFibonacci(n){
    if(n == 0 || n == 1){return n;}else{return slowFibonacci(n-1) + slowFibonacci(n-2);}
}
// memoized fibonacci
var fibonacci = (function() {
    var memo = {}; // our hash to memo
    function f(n) {
        var value;
        if (n in memo) {
            value = memo[n];
        } else {
            if (n === 0 || n === 1)
                value = n;
            else
                value = f(n - 1) + f(n - 2);
            memo[n] = value;
        }
        return value;
    }
    return f; // return memoized function
})();

console.time("slow");
console.log( "value", slowFibonacci(34));
console.timeEnd("slow");
// value 5702887
// slow: 10711ms (that's almost 11 seconds!!!)
// mileage may very, ran MUCH faster in chrome


console.time("memo");
console.log("value", fibonacci(34));
console.timeEnd("memo");
// value 5702887
// memo: 1ms
```

slowFibonacci



2*fib(n+1) -1 calls

Memoize Fibonacci



n + (n-1) worst case

Automatic Memoization

```
// function to convert a non-memoized function to memoized version
function memoize(func) {
    var memo = {};
    var slice = Array.prototype.slice;

    return function () {
        var args = slice.call(arguments); //convert args to array

        if (args in memo) // auto-magic conversion to string of args
            return memo[args];
        else
            return (memo[args] = func.apply(this, args));

    }
}

var memoFib = memoize(slowFibonacci);
```

Partial Application/Curry

```
// partially apply a set of arguments to a function
function partial() {
    var fn = arguments[0]; // use first argument as function
         // save rest of arguments for final call
    var args = Array.slice(arguments, 1);
return function () {
        // use previous args + any arguments from this call and apply them
         // to function
        return fn.apply(this, args.concat(Array.slice(arguments, 0)));
    };
}

function test(x, y) {
    return x + y;
}

var pTest = partial(test, 4);
pTest(6); // 10
```

Curry added to Function Prototype

```
// Same as partial, but called from function
// f.curry([args…])
Function.prototype.curry = function (/*args...*/) {
    var fn = this;
    var args = Array.slice(arguments, 0); // args to curry call
    return function () {
        return fn.apply(this, args.concat(Array.slice(arguments, 0)));
    };
}

function test(x, y, z) {
    return x + y + z;
}

var a = test.curry(4);
a(6, 5); // 15

var b = test.curry(5,5);
a(1); // 16
```

Curry

```
// an example with a function
function performOperation(oper, x, y) {
    return oper(x, y);
}

var mult = performOperation.curry(function (x,
y) { return x * y; });
var add = performOperation.curry(function (x,
y) { return x + y; });

mult(2, 3); // 6
add(2, 3); // 5
```

Function composition

```
// compose multiple functions into something like f(g(x))
function compose() {
var fns = arguments; // we need the list of our functions, put them in closure scope
    var arglen = fns.length;
    return function() {
        for (var i = arglen - 1; i >= 0; --i) {
            arguments = [fns[i].apply(this, arguments)]; // apply supplied args first
                    //time, or computed args each time after
        }
        return arguments[0]; // return our final computation
    };
}
function sq(x) { return x * x;}

function inc(x) {return x + 1;}


var test = compose(inc, sq);
test(3); // 10 = (3^2)+1 = inc(sq(3));
```

Function composition (cont.)

```
// like composition, but in reverse
function pipe() {
    var fns = arguments;
    var arglen = fns.length;
    return function() {
        for (var i = 0; i < arglen; i++) {
            arguments = [fns[i].apply(this, arguments)];
        }
        return arguments[0];
    };
}


function sq(x) { return x * x;}

function inc(x) {return x + 1;}


var test1 = pipe(inc, sq);
test1(3); //16 = (3+1)^2 = sq(inc(3));
```

Extra’s

- End of Presentation
- Beyond this point, are just a few “practical” examples

Using generically

- Many of javascript's higher order functions are written generically enough to be used on any Array-like object
- Natively, this means strings

```
// ASCII Byte Encoding
var map = Array.prototype.map;
var result = map.call("Hello world", function (x) {
return x.charCodeAt(0); });

// result = [72, 101, 108, 108, 111, 32, 119, 111,
114, 108, 100]
```

Using generically (cont.)

```
// map a DNA strand to it's complement
function complementDna(str) {
    var map = Array.prototype.map;
    return map.call(str, function (x) {
        if (x === "A") {
            return "T";
        } else if (x === "T") {
            return "A";
        } else if (x === "C") {
            return "G";
        } else if (x === "G") {
            return "C";
        } else {
            return x;
        }

    }).join("");
}

var result = complementDna("AAATCGCA");
// result = "TTTAGCGT"
```

Using generically (cont.)

```
//Get a frequency hash of all k-mers in data stream
function frequencyKmer(k, data) {
    var map = Array.prototype.map;
    var mapping = map.call(data, function (x, i) {
        // get the string starting at i, with a length of k
        if (i + k <= data.length) {
            return data.substr(i, k);
        }
    });

    return mapping.reduce(function (acc, word) {
        if (!word) {// we map last k-1 elements as undefined, ignore
        } else {
            acc[word] ? acc[word]++ : acc[word] = 1;
        }
        return acc;
    }, {});
}
frequencyKmer(4, "ACTGGTACTG");
// {ACTG: 2, CTGG: 1, GGTA: 1, GTAC: 1, TACT: 1, TGGT: 1}
```