Functional Programming in javascript

Note: Estimated time: 1h

What is functional programming?

A programming paradigm.

What is a programming paradigm?

> Paradigms are a distinct set of concepts or thought patterns.
>
> Programming paradigms are a way to classify programming languages according to the style of computer programming.

Other programming paradigms:

- Imperative programming
- Object-oriented programming
- and many more

Main concepts

- **First-class functions** | functions to be treated like any other value
- **Immutability** | don't mutate data
- **Absence of side effects** | don't alter state (stateless)
- **Lambda Calculus** | function expressions with closures
- **Recursion** | don't iterate (no loops)

First-class functions

> Functions to be treated like any other value. This means they can be created, passed to

> functions, returned from functions and stored inside data structures

---

Functions are objects

```
var logToConsole = function (text) {
  console.log(text)
}

console.log(typeof logToConsole);
// object
```

Function at MDN

Functions are objects

```
var logToConsole = function (text) {
  console.log(text)
}
```

Is the same as:

```
var logToConsole = new Function('text', 'console.log(text)');
```

As objects, they can have properties, methods, etc.

```
logToConsole.count = 1
console.log(logToConsole.count);
// 1
```

Functions can be stored as variables

```
var logToConsole = function (text) {
  console.log(text)
```

```
}

var log = logToConsole
log('Hello World')
```

Functions can be passed as parameter

```
var numbers = [1, 4, 9];
var doubles = numbers.map(function(num) {
  return num * 2;
});
```

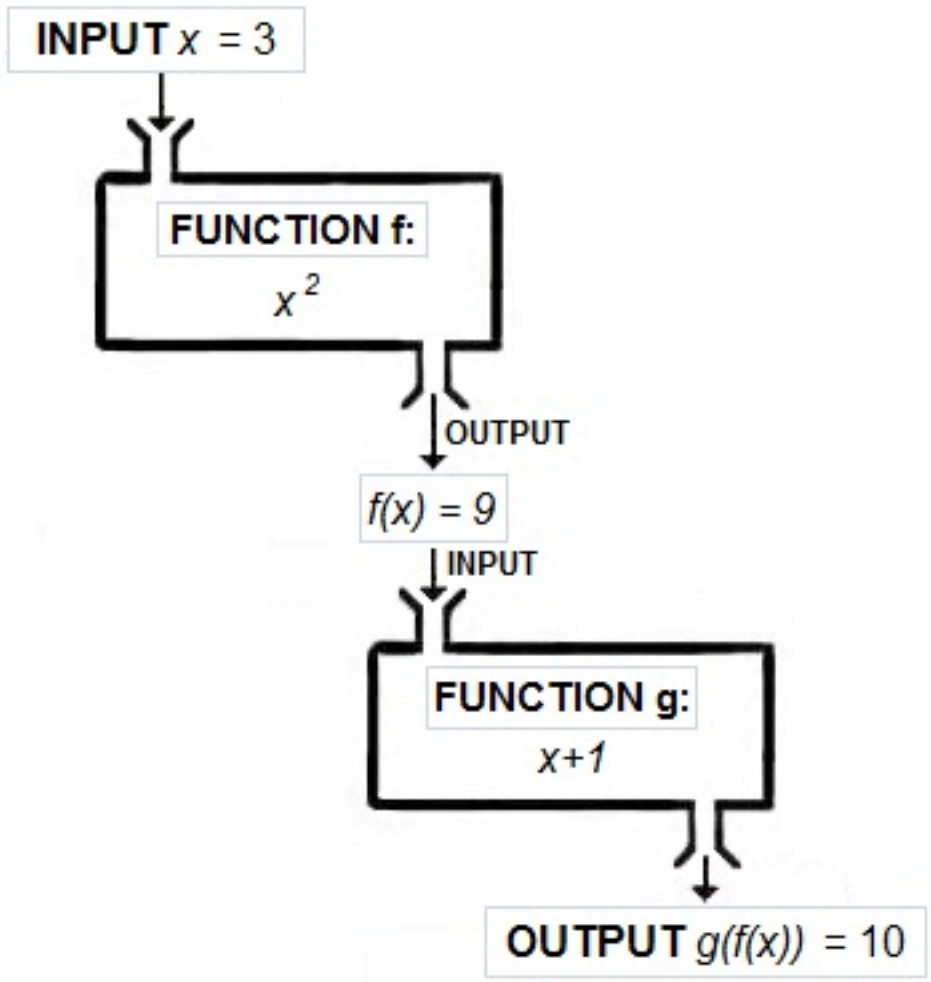
Functions can be returned

```
var createIncrementer = function(increment) {
  return function( number ) { return number + increment }
}

var increment3 = createIncrementer(3)
increment3(5)
// 8
```

Immutability

> An immutable object is an object whose state cannot be modified after it is created.
>
> A function is not supposed to mutate its input(s) but to return a newly created output



Data mutation in a function

```
function increment1 (numbers) {
  for (var i in numbers) {
    numbers[i]++
  }
  return numbers
}
var numbers = [4,5,6]
var incrementedNumbers = increment1(numbers)

console.log(numbers) // [5, 6, 7]
console.log(incrementedNumbers) // [5, 6, 7]
```

Note: in this function, `numbers` is mutated by `increment1`

Without mutation

```
function increment1 (numbers) {
  var result = []
  for (var i in numbers) {
    result[i] = numbers[i] + 1
  }
  return result
}
var numbers = [4,5,6]
var incrementedNumbers = increment1(numbers)

console.log(numbers) // [4, 5, 6]
console.log(incrementedNumbers) // [5, 6, 7]
```

Note: in this function, `numbers` is not mutated and an output array is created instead.

Lambdas

> In computer science, the most important, defining characteristic of a lambda expression is that it is used as data.

- Passed as an argument to another function to be invoked
- Function returned as a value from a function
- Assigned to variables or data structures

Note: In JavaScript, not all lambdas are anonymous, and not all anonymous functions are lambdas, so the distinction has some practical meaning.

> λx.x*x

```
function square(x) {
  return x * x;
}
var squares = [3,4,6].map( square )
```

```
var squares = [3,4,6].map( x => x*x )
```

Note: Both `square` functions can be considered lambdas, but the first one would be the more pure as expressed as an expression

---

> λx.λ.y.x+y

```
function plus(x,y) {
  return x + y;
}
plus(5,7);
```

> λx.λ.y.x+y

becomes

```
function plus(x) {
  return function plusx(y) {
    return x + y;
  }
}
plus(5)(7)
```

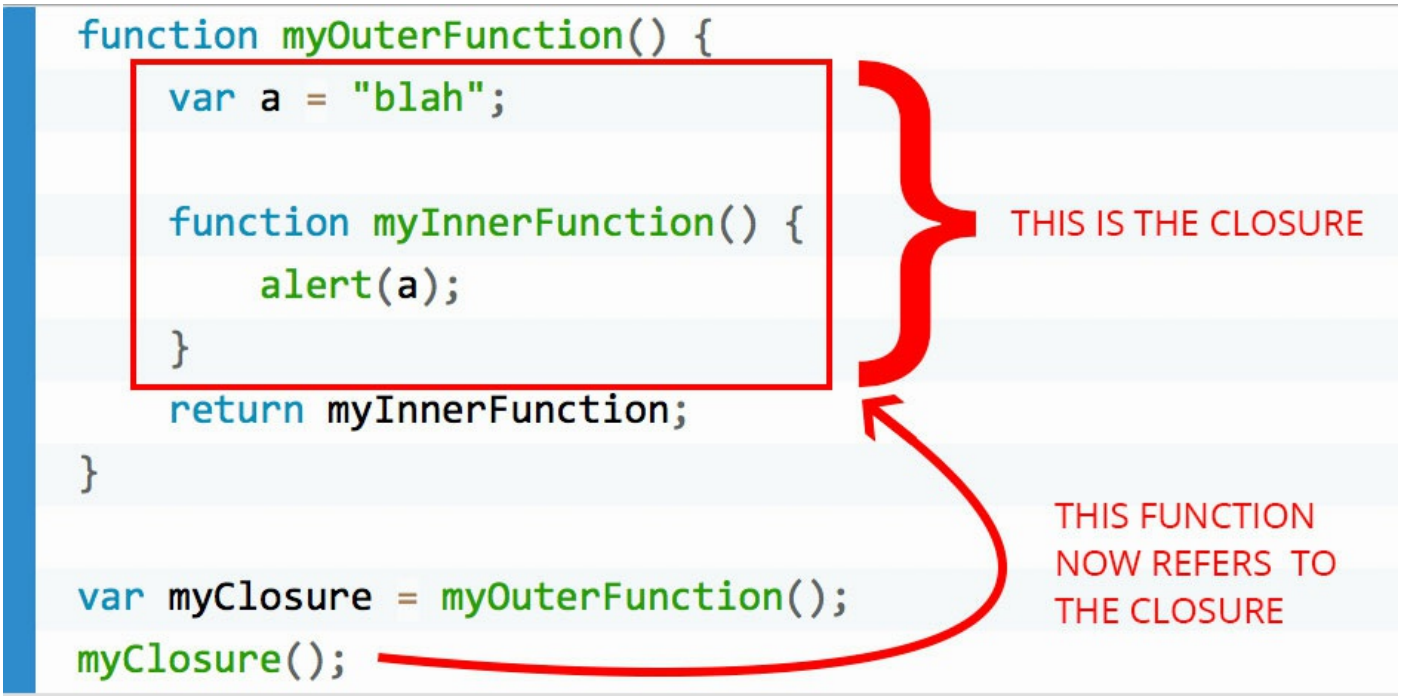
becomes

```
(x => y => x + y)(5)(7)
```

Note: `plus` is a lambda expression, even if it´s not anonymous.

Closures

> Lambdas are possible in javascript thanks to closures.
>
> A closure is a special kind of object that combines two things: a function, and the environment in which that function was created.




```
function makeFunc() {
  var name = "Mozilla";
  function displayName() {
    alert(name);
  }
  return displayName;
}

var myFunc = makeFunc();
myFunc();
`
```

Note: When `displayName` is created, `name` is added to its closure with its value 'Mozilla'

```
function makeAdder(x) {
  return function add(y) {
    return x + y;
  };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2));  // 7
console.log(add10(2)); // 12
```

https://jsbin.com/jegerel/edit?js,console

Note: When `add` is created, `x` is added to its closure with its current value. First with `x=5` and then with `x=10` That´s why, `add5` and `add10` return different values as they have a different `x` in their closure.

Absence of side effects

> It doesn't rely on data outside the current function
>
> It doesn't change data that exists outside the current function.

Statelessness

> The state is a snapshot of a program's current environment: all the variables that have been declared, functions created and what is currently being executed.
>
> An expression in a programming language can be "stateful" or "stateless". A stateful expression is one that changes a program's current environment.

Don´t change state

This is not functional:

```
var number = 1;
var increment = function() {
    return number += 1;
};
increment();
```

This is a functional:

```
var number = 1;
var increment = function(n) {
    return n + 1;
};
increment(number);
```

Don´t depend on state

This is not functional:

```
var birthday = new Date("December 17, 1995 03:24:00")

function isMillennial() {
  return birthday.getFullYear() > 1995
}
```

This is a functional:

```
function isMillennial(birthday) {
  return birthday.getFullYear() > 1995
}
```

Recursion

> As opposed to iteration, method where the solution to a problem depends on solutions to smaller instances of the same problem
>
> A recursive function, as you saw in CS100, is one that calls itself

---

Don't iterate

> Use recursion functions instead: map, reduce, filter, etc...

Imperative style

```
var numbers = [1,3,5,6]
var squares = []
for( var i; i < numbers.length; i++) {
  squares = numbers[i] * numbers[i]
}
```

Functional style

```
var numbers = [1,3,5,6]
var squares = numbers.map( function (num) {
  return num * num
} )
```

More about functions

Pure functions

- Giving same input will always return same output
- Produces no side effect: user output, memory writing, logging...
- Does not mutate input

Impure functions

- Relies on current time
- Random numbers
- Side effects
- Mutation

Higher-order functions

> A higher-order function is a function that does at least one of the following:
>
> - Take one or more functions as an input
> - Output a function
>
> All other functions are first order functions.

```
function negate(func) {
  return function(x) {
    return !func(x);
  };
}
var isNotNaN = negate(isNaN);
show(isNotNaN(NaN));
```

map()

> The `map()` method creates a new array with the results of calling a provided function on every element in this array.

```
var numbers = [1, 4, 9];
var doubles = numbers.map(function(num) {
  return num * 2;
});
// doubles is now [2, 8, 18]. numbers is still [1, 4, 9]
```

reduce()

> The `reduce()` method applies a function against an accumulator and each value of the array (from left-to-right) to reduce it to a single value.

```
var result = [0,1,2,3,4].reduce( (previousValue, currentValue, currentIndex, array) => {
  return previousValue + currentValue;
}, 10);
// 20
```

```
var flattened = [[0, 1], [2, 3], [4, 5]].reduce(function(a, b) {
  return a.concat(b);
}, []);
// flattened is [0, 1, 2, 3, 4, 5]
```

Is Javascript really functional?

It is

- Functions are first class objects
- Lambdas
- Closures (encapsulate the state of the function)

- Higher order functions (pass functions to functions)

But it can also be Imperative

> Sequence of steps/instructions that happen in order and modifies the state

- while statements
- for loops
- ...

And it's also Object-oriented

> Concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.

- Everything is an object
- Native objects (and arrays) are mutable by default
- Prototype inheritance
- ES6 introduces "real" classes

Javascript is a multi-paradigm programming language

Imperative style: `loops`

```
function simpleJoin(stringArray) {
 var accumulator = ''
 for (var i=0, l=stringArray.length; i < l; i++) {
   accumulator = accumulator + stringArray[i]
 }
 return accumulator
}
```

Note: program state change is achieved by executing a series of statements, and does flow control primarily using conditional statements, loop statements and function calls.

Functional style: `recursion`

```
function simpleJoin(stringArray, i, accumulator) {
 if (i === stringArray.length) {
   return accumulator
 } else {
   return simpleJoin(stringArray, i+1, accumulator+stringArray[i])
 }
}
```

https://jsbin.com/mocaya/edit?js,console,output

Note: does not have any loop statements. Instead it uses recursion for iteration.

And even more functional:

```
function simpleJoin(stringArray, i, accumulator) {
 return (i === stringArray.length) ? accumulator
   : simpleJoin(stringArray, i+1, accumulator+stringArray[i])
}
simpleJoin(['a', 'b', 'c'] , 0, '')
```

Note: does not have any `if`. It uses an expression that evaluate to some value, instead of statements that don't evaluate to anything.

Taking advantage of ES6:

```
function simpleJoin(stringArray, i = 0, accumulator = '') {
 return (i === stringArray.length) ? accumulator
   : simpleJoin(stringArray, i+1, accumulator+stringArray[i])
}

simpleJoin(['a', 'b', 'c'])
```

Object-oriented style: `as method`

```
Array.prototype.simpleJoin = function() {
 var accumulator = “”
 for (var i=0, l=this.length; i < l; i++) {
   accumulator = accumulator + this[i]
 }
 return accumulator
```

```
}
```

Note: Object oriented languages tend to be imperative languages also. In this case the statements act on array object, not a given array.

Functional libraries for javascript

- underscore-js
- functional-js
- ramda-js
- lazy-js
- etc

Questions

Recursion

Should we choose recursion over loops?

> "Recursion is not intrinsically better or worse than loops - each has advantages and disadvantages, and those even depend on the programming language (and implementation)."
>
> "Iterative loops require destructive state updates, which makes them incompatible with pure (side-effect free) language semantics"

Source (stackoferflow)

> "The problem with recursion is that it (usually) uses more memory, a lot more. That's because each active call to a function is stored on what's called a call stack."

Watch animated explanation

> "Use recursion when it's the easier option (and N won't be that large)."

Source (reddit)

> "In my opinion, recursive algorithms are a natural fit when the data structure is also recursive."

Source (stackoverflow)

> "Practically speaking, if you're not using recursion for the following (even in imperative languages) you're a little mad:
>
> - Tree traversal
> - Graphs
> - Lexing/Parsing
> - Sorting"

Source (stackoverflow)

Traverse with recursion

```
function traverse (current, depth) {
  var children = current.childNodes
  for (var i = 0, len = children.length; i < len; i++) {
    // DO STUFF
    traverse(children[i], depth + 1)
  }
}
```

Traverse with iteration

```
function traverse (current) {
  var stack = [current]
  var stackIdx = 0
  var children, i, len

  while (current = stack[stackIdx++]) {
    children = current.childNodes
    for (i = 0, len = children.length; i < len; i++) {
      // DO STUFF
      stack.push(children[i])
    }
  }
```

```
}
```

Let's see if what we've read is true.

https://jsbin.com/qajegal/edit?js,console

Practice

Exercise: print array

> Print, with one line of code, the same output printed with `printArray`

```
var names = ["Ben", "Jafar", "Matt"];

var printArray = function(names, console) {
  var counter;

  for(counter = 0; counter < names.length; counter++) {
    console.log(names[counter]);
  }
}

printArray(names, console) // OUTPUT: "Ben" "Jafar" "Matt"
```

https://jsbin.com/kokoqe/edit?js,console

Solution

```
names.forEach( (x) => console.log(x) )
```

Note: Why not using map? Map is to created to return a new array. We don´t need any return here.

Exercise: map implementation

> Implement `map2` function to obtain same output as `map`

- Do not use any for/while loops.

```
Array.prototype.map2 = function(func) {
  // SOLUTION GOES HERE
};

var result =
console.log(
  [1,2,3].map2(function(x) { return x + 1; }),
  [1,2,3].map(function(x) { return x + 1; })
) // OUTPUT: [2, 3, 4] [2, 3, 4]
```

https://jsbin.com/yaqite/edit?js,console

Solution

```
Array.prototype.map2 = function(func) {
  var results = [];
  this.forEach(function(itemInArray) {
    results.push(func(itemInArray));
  });
  return results;
};
```

Exercise: chaining

> Create array with ids of videos that have a rating of 5.0

- Use only Array# higher-order methods

```
var videos = [{...},, ...];

function getBestVideosIds (videos) {
  // SOLUTION GOES HERE
}

console.log(getBestVideosIds(videos)) // OUTPUT [654356453, 675465]
```

https://jsbin.com/junawu/edit?js,console

Solution

```
function getBestVideosIds (videos) {
  return videos.filter(function(video) {
    return video.rating === 5.0;
  })
  .map(function(video) {
    return video.id;
  })
}
```

Exercise: recursion

> Implement a function that takes a function as its first argument, a number num as its second argument, then executes the passed in function num times.

```
function repeat(operation, num) {
  // SOLUTION GOES HERE
}

repeat( () => console.log(1), 7 )
// OUTPUT: 1 1 1 1 1 1 1
```

https://jsbin.com/joneliv/edit?js,console

Solution

```
function repeat(operation, num) {
  if (num <= 0) return
  operation()
  return repeat(operation, --num)
}
```

Exercise: immutability

> Replace `cloneDeep` to avoid data mutation

- Loops are not allowed

```
var data = { /* ... */ }

function clone(data) {
  // YOUR CODE GOES HERE
  return data;
}

clonedData = clone(data)
clonedData.users[2].name = 'Fake user name'
clonedData.users[0].games[0].name = 'Fake game name'
```

https://jsbin.com/kulufo/edit?js,console,output

Solution

```
function clone(data) {
  var result;

  if (isArray(data)) {
    result = data.map( child => clone( child ) )
  } else if (typeof data == "object") {
    result = {}
    Object.keys(data).forEach(function(i){
      result[i] = clone( data[i] );
    })
  } else {
    result = data;
  }

  return result;
}
```

Exercise: reduce

> Write a function `countZeroes`, which takes an array of numbers as its argument and returns the amount of zeroes that occur in it. Use `Array#reduce`.

```
function countZeroes(array) {
  // SOLUTION GOES HERE
}
```

```
console.log(
  countZeroes([1,2,0,0,4,1,0,2,0,1])
)

// OUTPUT: 4
```

https://jsbin.com/cubidal/edit?js,console

Solution

```
function countZeroes(array) {
  function counter(total, element) {
    return total + (element === 0 ? 1 : 0);
  }
  return array.reduce(counter, 0);
}
```

Exercise: abstraction

> Write the higher-order function `count`, which takes an array and a test function as arguments, and returns the amount of elements in the array for which the test function returned true. Re-implement `countZeroes` using this function.

```
console.log(
  countZeroes([1,2,0,0,4,1,0,2,0,1])
)
// OUTPUT: 4
```

https://jsbin.com/sotepiq/edit?js,console

Solution

```
function count(test, array) {
  return array.reduce(function(total, element) {
    return total + (test(element) ? 1 : 0);
  }, 0 );
}

function equals(x) {
  return function(element) {return x === element;};
```

```
}

function countZeroes(array) {
  return count(equals(0), array);
}
```

Note: now the code is pure and much more functional and reusable; each function does a single thing.

The end

Apply your new knowledge to

/github/js-training-practice/functional-programming