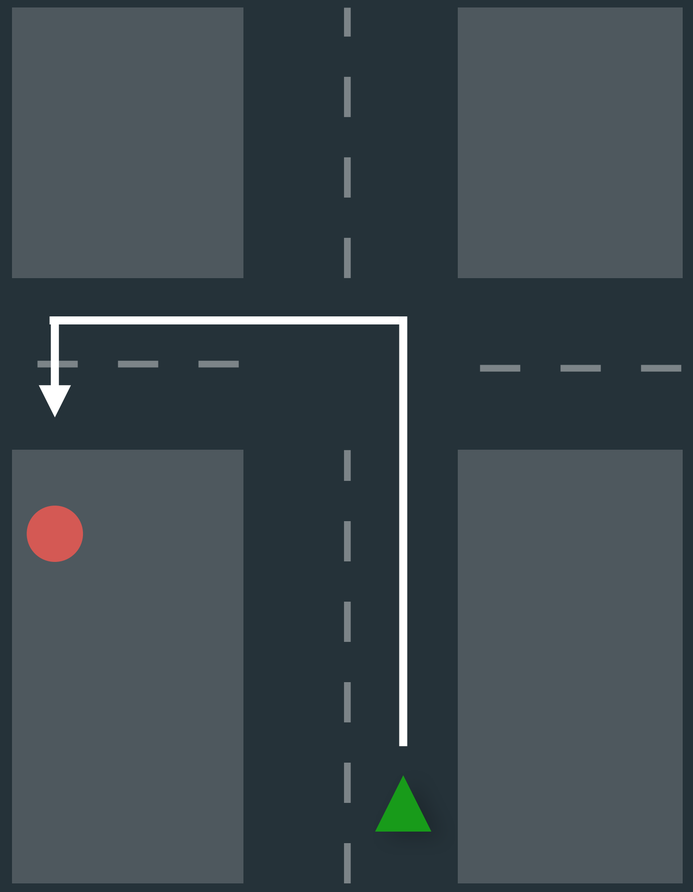
Higher Order Functions

By Minh for FCC Toronto

The Problem

Where is the market?



- **A simple answer**
  1) Go around the corner to the left

- **A good answer**
  1) Go straight
  2) Turn left

- **A really really really unnecessarily complicated answer:**
  1) Open your car door
  2) Get in the car
  3) Close the car door
  4) Turn on the engine
  5) …

Abstractions

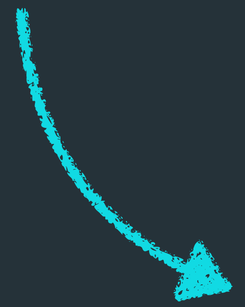
> In the context of programming, these kinds of vocabularies are usually called abstractions. Abstractions **hide details** and give us the ability to **talk about problems at a higher (or more abstract) level.**

Eloquent JavaScript:
A Modern Introduction to Programming

But how do we do this?

The answer: Functions

> “In computer programming, a subroutine (a.k.a function) is a sequence of program instructions that perform a specific task, packaged as a unit.”
>
> Wikipedia, 2017



Can be “refactored”*

*Code refactoring: the process of restructuring existing computer code—changing the factoring—without changing its external behavior.

The birth of a higher order

```
1  var numArray = [1, 2, 3];
2  var stringArray = ["dog", "cat", "lion"];
3  var objArray = [ {type: "dog"}, {type: "cat"}, {type: "cat"} ];
```

Project manager: “we need to log out each content of **numArray**!”

```
7  for (var i = 0; i < numArray.length; i++) {
8    var current = numArray[i];
9    console.log(current);
10 }
```

Day 1

Project manager *: “we also need to log out each content of **numArray**, **stringArray** and **objArray**!”

```
14 function logEach(array) {
15   for (var i = 0; i < array.length; i++) {
16     var current = array[i];
17     console.log(current);
18   }
19 }
20 logEach(numArray);
21 logEach(stringArray);
22 logEach(objArray);
```

Day 2

Project manager: “we need to do **3 other** things to each content of **numArray**, **stringArray** and **objArray**!”

```
var codeFitsSlide = false;
```

¯_(ツ)_/¯

Day 3

Birth of a higher order function

SOLUTION 1

```
function doSomething1toEach(array){
  for (var i = 0; i < array.length; i++) {
    var current = array[i];
    doSomething1(current);
  }
}
function doSomething2toEach(array){
  for (var i = 0; i < array.length; i++) {
    var current = array[i];
    doSomething2(current);
  }
}
function doSomething3toEach(array){
  for (var i = 0; i < array.length; i++) {
    var current = array[i];
    doSomething3(current);
  }
}
doSomething1toEach(numArray);
doSomething1toEach(stringArray);
doSomething1toEach(objArray);
                ...
```

SOLUTION 2

```
function forEach(array, callBack) {
  for (var i = 0; i < array.length; i++) {
    var current = array[i];
    callBack(current);
  }
}
forEach(numArray,    doSomething1toEach);
forEach(stringArray, doSomething1toEach);
forEach(objArray,    doSomething1toEach);
forEach(numArray,    doSomething2toEach);
forEach(stringArray, doSomething2toEach);
forEach(objArray,    doSomething2toEach);
forEach(numArray,    doSomething3toEach);
forEach(stringArray, doSomething3toEach);
forEach(objArray,    doSomething3toEach);
```

Higher-Order Function

Even better

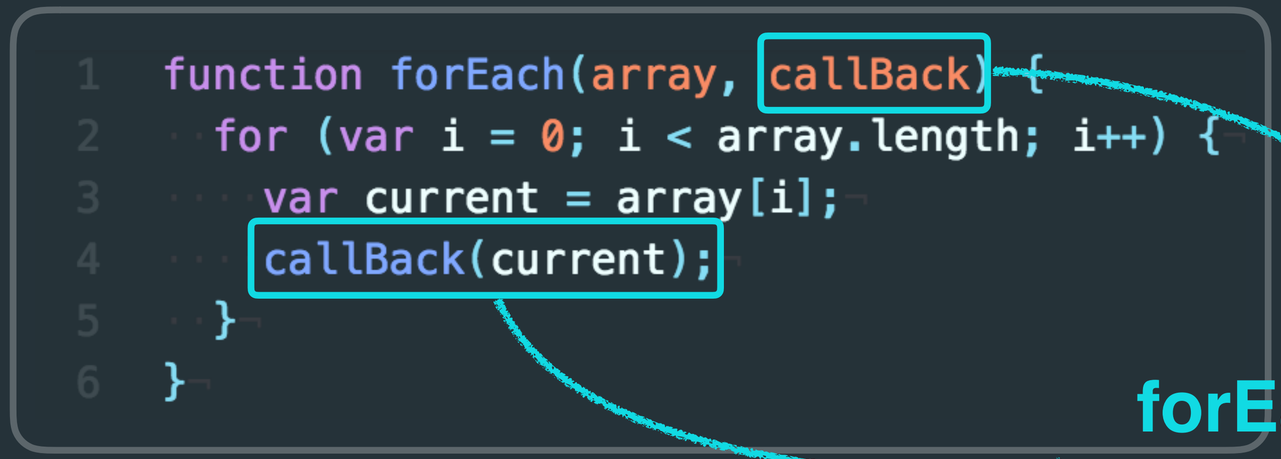
SOLUTION 3 - Unleash the higher power!!!

```
function forEach(array, callBack) {
  for (var i = 0; i < array.length; i++) {
    var current = array[i];
    callBack(current);
  }
}
forEach([numArray, stringArray, objArray], function(array){
  forEach(array, function(element){
    forEach([doSomething1, doSomething2, doSomething3], function(myFunc){
      myFunc(element)
    })
  })
})
```

Higher-Order Functions

> **Functions that operate on other functions**, either by taking them as arguments or by returning them, are called higher-order functions.

Eloquent JavaScript:
A Modern Introduction to Programming



forEach() operates on another function received as argument

Nerd Fact

So nerdy
I can't even…

$$y = f(x) = a(x)^2 + b(x) + c$$

$$\frac{d}{dx}(y) = f'(x) = 2(a)(x) + b$$

The Differential Operator in calculus is a **mathematical "Higher-Order Function"** because it is a function which **receives a function** (*f(x)*) **and returns another** function (*the derivative of f(x)*)

Commonly used Javascript HOFs - filter

`Array.prototype.filter`

Each element of the array is passed into a callback function, which **returns true if the element passes the test** specified by the callback, **false otherwise**. Filter **returns a new array** containing **elements of the original array where the callback returned true.**

Syntax

```
[1, 2, 3, 4, 5].filter(function(num, index, array){
  return num % 2 === 1;
}, thisValue)
```

```
[1, 2, 3, 4, 5].filter(function(num){
  return num % 2 === 1;
})
```

```
[1, 2, 3, 4, 5].filter(num => num % 2 === 1)
```

Parameters

- **callback:** function that contains the test logic. receives three arguments:
  - **element:** The current element being processed in the array.
  - **index:** The index of the current element being processed in the array.
  - **array:** The original array filter was called upon.
- **thisArg:** Optional. Value to use as this when executing callback.

Commonly used Javascript HOFs - map

`Array.prototype.map`

Each element of the array is passed into a callback function, which **returns a transformed version of the original element**, given the transformation provided by the callback. Map **returns a new array that contains all the transformed version** of the elements of the original array.

Syntax

```
[1, 2, 3, 4, 5].map(function(num, index, array){
  return num % 2 === 1;
}, thisValue)

---

[1, 2, 3, 4, 5].map(function(num){
  return num % 2 === 1;
})

---

[1, 2, 3, 4, 5].map( num => num % 2 )
```

Parameters

- **callback:** function that contains the transformation logic. receives three arguments:
  - **element:** The current element being processed in the array.
  - **index:** The index of the current element being processed in the array.
  - **array:** The original array filter was called upon.
- **thisArg:** Optional. Value to use as this when executing callback.

Commonly used Javascript HOFs - reduce

`Array.prototype.reduce`

Applies a function against an accumulator and each element in the array to reduce it to one value.

Syntax

```
[1, 2, 3, 4, 5].reduce(function(sum, num, index, array){
  return sum + num;
}, 0)

---

[1, 2, 3, 4, 5].reduce(function(sum, num){
  return sum + num;
}, 0)

---

[1, 2, 3, 4, 5].reduce((sum, num) => sum + num)
```

Parameters

- **callback:** function that applies the accumulation logic. Receives 4 arguments:
  - **accumulator:** The thing that gets accumulated with every pass.
  - **element:** The current element being processed in the array.
  - **index:** The index of the current element being processed in the array.
  - **array:** The original array filter was called upon.
- **thisArg:** Optional. Value to use as this when executing callback.

Quiz: What do these give us?

map

```
[1, 2, 3, 4, 5]
  .map( num => num % 2 === 1)
```

filter

```
[1, 2, 3, 4, 5]
  .filter(num => num % 2 === 1)
```

reduce

```
[1, 2, 3, 4, 5]
  .reduce((sum, num) => sum + num)
```

Commonly used Javascript HOFs - Summary

map

```
[1, 2, 3, 4, 5]
  .map( num => num % 2 === 1)
```

```
[true, false, true, false, true]
```

Return format:
- Array of same length
- Different content type

filter

```
[1, 2, 3, 4, 5]
  .filter(num => num % 2 === 1)
```

```
[1, 3, 5]
```

Return format:
- Array of smaller length
- Same content type

reduce

```
[1, 2, 3, 4, 5]
  .reduce((sum, num) => sum + num)
```

```
15
```

Return format:
- Single value

Array traversal - Reduce is all you need!

```
3  function filter (array, callback) {
4    var output = [];
5    for(var i = 0; i < array.length; i++){
6      if(callBack(array[i]) === true){
7        output.push(array[i])
8      }
9    }
10   return output;
11 }
12
13 function map (array, callback) {
14   var output = [];
15   for(var i = 0; i < array.length; i++){
16     var newValue = callback(array[i]);
17     output.push(newValue);
18   }
19   return output;
20 }
```

Accumulator!

```
[1, 2, 3, 4, 5]
  .map( num => num % 2 === 1)
```

```
[1, 2, 3, 4, 5]
  .reudce((accum, num) => {
    accum.push( num % 2 === 1 )
    return accum;
  },[])
```

```
[
  true,
  false,
  true,
  false,
  true
]
```

```
[1, 2, 3, 4, 5]
  .filter(num => num % 2 === 1)
```

```
[1, 2, 3, 4, 5]
  .reudce((accum, num) => {
    if( num % 2 === 1 ){
      accum.push( num )
    }
    return accum;
  },[])
```

```
[1, 3, 5]
```

Why use HOF?

```
var dragonList = [
  {name: "Justin", age: 3000, canFly: true},
  {name: "Andrew", age: 2,    canFly: false},
  {name: "Minh",   age: 240,  canFly: true},
  {name: "Alex",   age: 250,  canFly: true},
]
```

```
function getDragonsWhoCanFly(){
  for(var i = 0; i < dragonList; i++){
    if(dragonList[i].canFly){
      dragonsWhoCanFly.push(dragonList[i]);
    }
  }
  return dragonsWhoCanFly;
}
```

Imperative

```
function getDragonsWhoCanFly(){
  return dragonList.filter(function(dragon){
    return dragon.canFly;
  })
}
```

Declarative

Why use HOF?

```
function getNameOfDragonsThatCanFly(){
  var nameOfDragonsThatCanFly = [];
  for(var i = 0; i < dragonList; i++){
    if(dragonList[i].canFly){
      nameOfDragonsThatCanFly.push(dragonList[i].name);
    }
  }
  return nameOfDragonsThatCanFly;
}
```

Imperative

- Too much distraction
- Hard to reason about
- Hard to debug

```
function getNameOfDragonsThatCanFly(){
  return dragonList.filter(function(dragon){
    return dragon.canFly;
  }).map(function(dragon){
    return dragon.name;
  })
}
```

Declarative

- Focuses on the big picture
- Easier to reason about
- Easier to debug

Array traversal - Other HOF

Array.prototype.forEach

Array.prototype.some

Array.prototype.every

Array.prototype.find

Array.prototype.findIndex

Curried function - another type of HOF

```
63  function isGreaterThan(a){
64    return function(b){
65      return b > a;
66    }
67  }
68  function getNamesOfDragonOlderThan200(){
69    return dragonList
70      .map(dragon => dragon.age)
71      .filter(isGreaterThan(200))
72      .map(age => dragonList.filter(dragon => dragon.age === age)[0] )
73      .map(dragon => dragon.name)
74  }
```

Quiz: What does this give us?

```
var dragonList = [
  {name: "Justin", age: 3000, canFly: true},
  {name: "Andrew", age: 2,    canFly: false},
  {name: "Minh",   age: 240,  canFly: true},
  {name: "Alex",   age: 250,  canFly: true},
]
```

```
dragonList.reduce(function(lastDragon, dragon){
  return dragon.age + lastDragon.age
})
```

Quiz: The answer

```
dragonList.reduce(function(lastDragon, dragon){
  return dragon.age + lastDragon.age
})
```

NaN

```
dragonList.reduce(function(totalAge, dragon){
  return totalAge + dragon.age;
}, 0);
```

3992

Dragon Challenge Set!

```
var dragonList = [
  {name: "Justin", age: 3000, canFly: true},
  {name: "Andrew", age: 2,    canFly: false},
  {name: "Minh",   age: 240,  canFly: true},
  {name: "Alex",   age: 250,  canFly: true},
]
```

1. Write a function that returns the names of the dragons who can fly
2. Write a function that returns the oldest dragon's age
3. Write a function that returns the second oldest dragon's age
4. Write a function that returns the name of the second oldest dragon