SAY WHAT NOT HOW

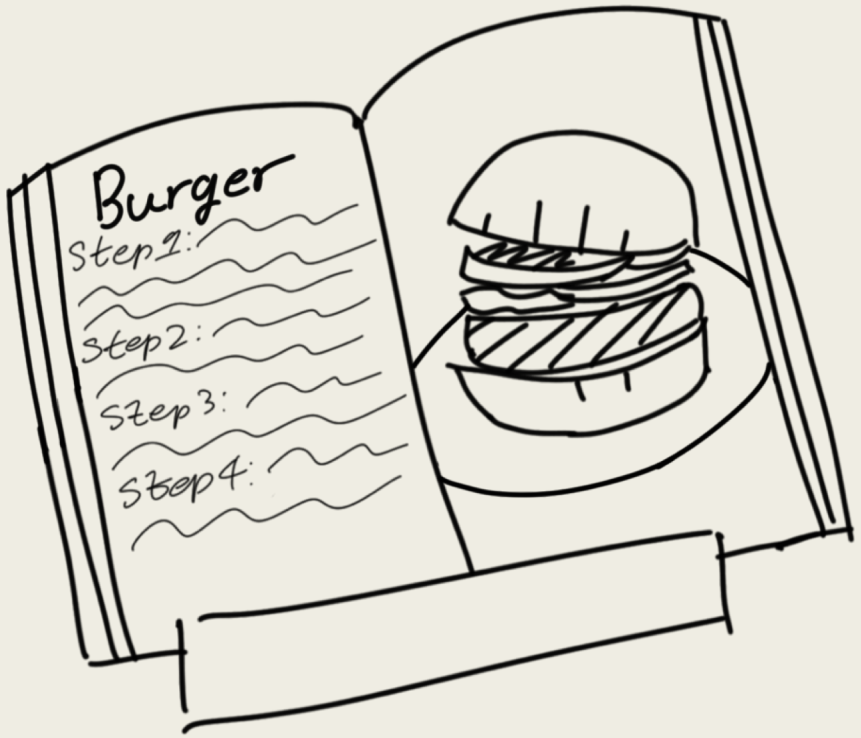
Introduction to Functional Programming

Agenda
What's In, What's Out

- I (and many others) started with Imperative programming; Functional programming is a kind of Declarative programming – so what's what?
- Focus on higher order functions – a grounded and common way to do functional programming. (We will ignore the kind that return functions)
- Avoiding everything else; I'm not qualified on that, and we only have so much time

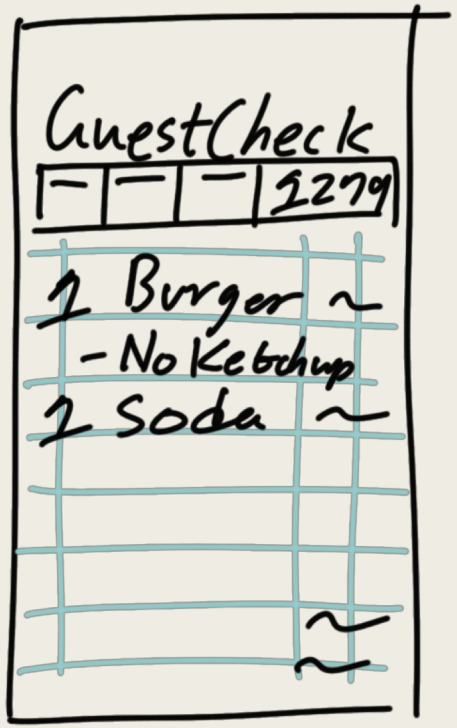
Imperative

How To Do It



Declarative

What To Do



Imperative Example
Say How to Do It

- Create a new empty list to hold the result (lets call it result list)
- Accept the list of all customers and iterate through the list.
  - *For each customer, check for these two conditions:*
    - Check their address and verify they live in Kansas City
    - Check if the customer has spent over 100 dollars
  - *If the above criteria is met, add the customer to the result list*
  - *If the above criteria is not met, then do not add the customer to the result list*
- After going through all customers, return the the result list

Imperative Example
Say How to Do It

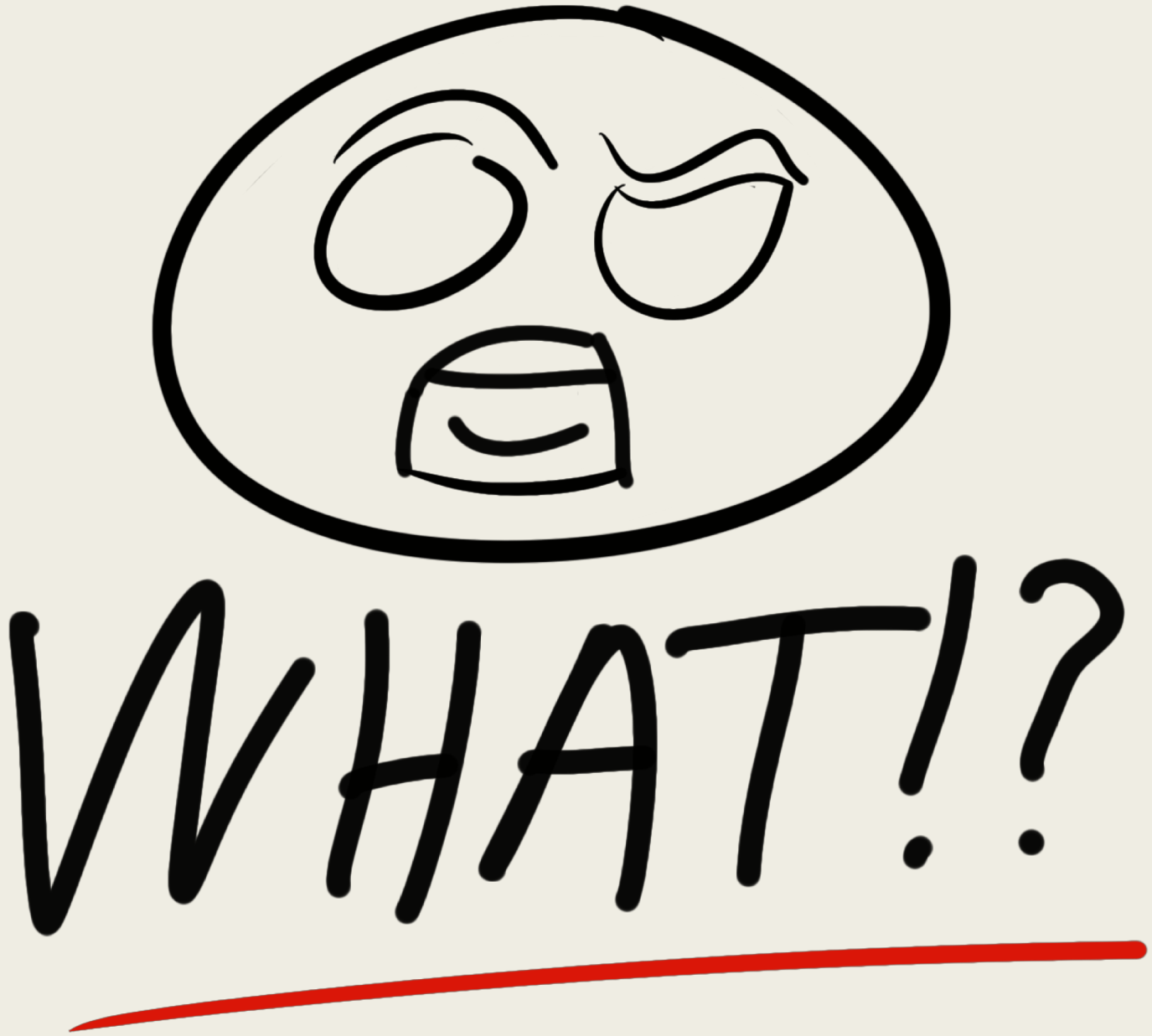
```
function CustomerReport(customers, city, amountSpentRequired){
    let result = [];
    for(let i = 0; i < customers.length; i++){
        let customer = customers[i];
        if(customer.address.city === city &&
            customer.amountSpent >= amountSpentRequired)
            result.push(customer)
    }
    return result;
}

let resultI = CustomerReport(allCustomers, 'KC', 100);
```

Declarative Example
Say What To Do

- I want all customers that live in Kansas City and have spent over 100 dollars.

```
let result = allCustomers
   .filter(x => x.address.city === 'KC' && x.amountSpent > 100 )
```


WHAT!?

.Filter? What?
Yo Dog, I Heard You Like Functions

- Filter is a Higher Order function in JavaScript
  - *For each element in the array it is called on, it executes the function passed in; if true it saves element: if not, discards the element.**
- So we don't worry about looping, storing intermediary results, or returning anything; we just focus on the logic we do care about.

*Technically a brand new list is created, because immutability

But what about “x =>”
Nothing more than shorthand

- In JavaScript, doing “x =>” is just shorthand for declaring an anonymous function (called fat arrow syntax)

```
allCustomers.filter(
   function(customer)
   {
      return customer.address.city === ‘KC’ &&
          customer.amountSpent > 100
   }
)
```

But Why Fat Arrow Syntax?
Complexity over simplicity?

- In my experience, many JavaScript examples use the fat arrow syntax over longhand syntax.
- Opinion: When I started, it felt intimidating to look at. Once I got it, I felt it it was more readable, reducing clutter and expressing the code I cared about

But why care about readable code?
What does ‘expressive’ code mean?

“Indeed, the ratio of time spent **reading versus writing is well over 10 to 1.** We are constantly reading old code as part of the effort to write new code. …[Therefore,] making it easy to read makes it easier to write.”

- Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship

Map, Reduce, Filter
The Trifecta

- Map
- Filter
- Reduce

Not just built in functions – key higher order functions found in lots of languages

Filter (JS)

4'3" To Ride

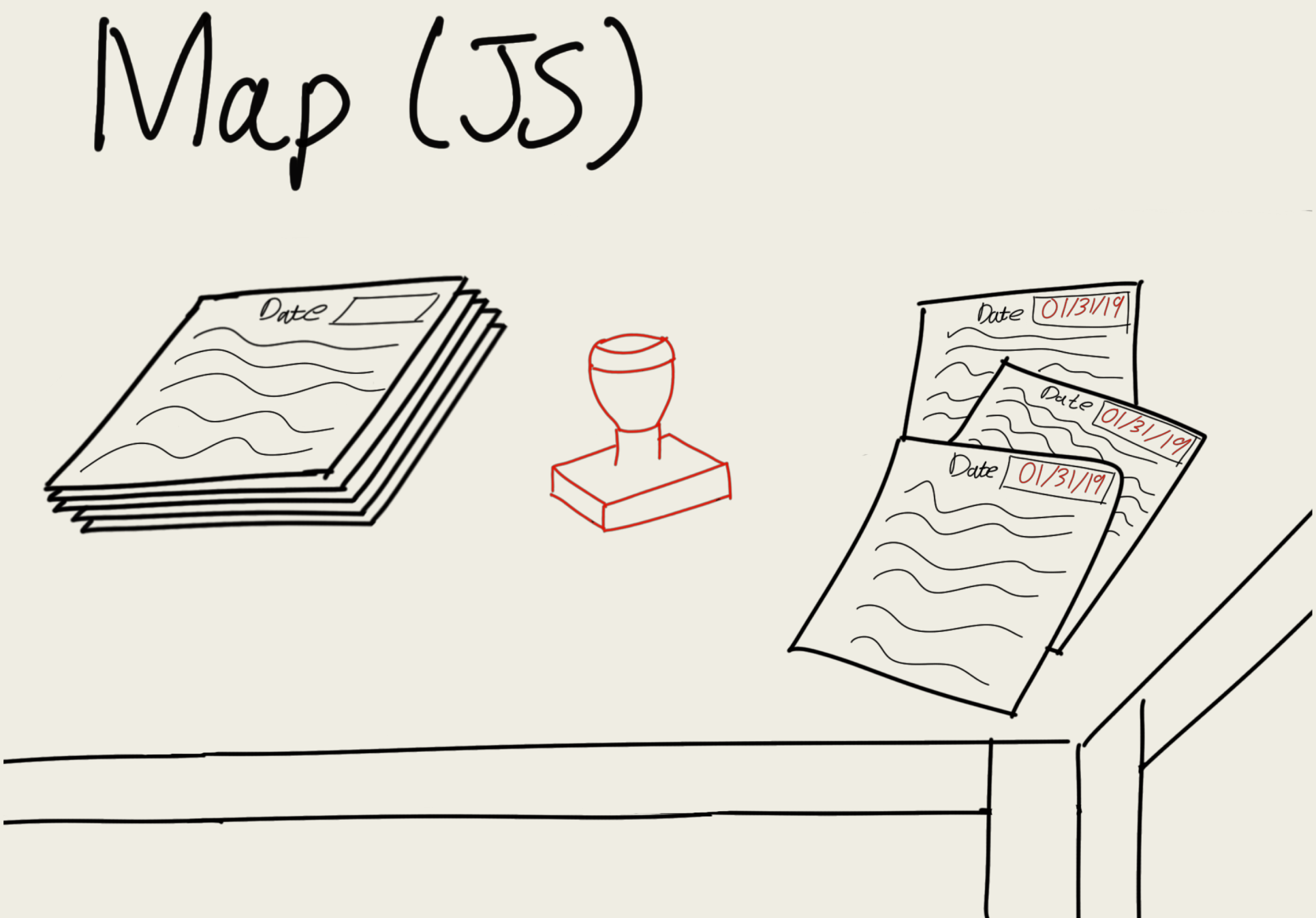
Filter Operator

Use when you want to pick data

```
function onlyTallEnough(heights, minHeight){
   let result = [];
   for(let i = 0; i < heights.length; i++){
      let height = heights[i]
      if(height >= minHeight)
         result.push(height)
   }
   return result;
}

let result = heights.filter(x => x >= 4.25);
```

Map (JS)



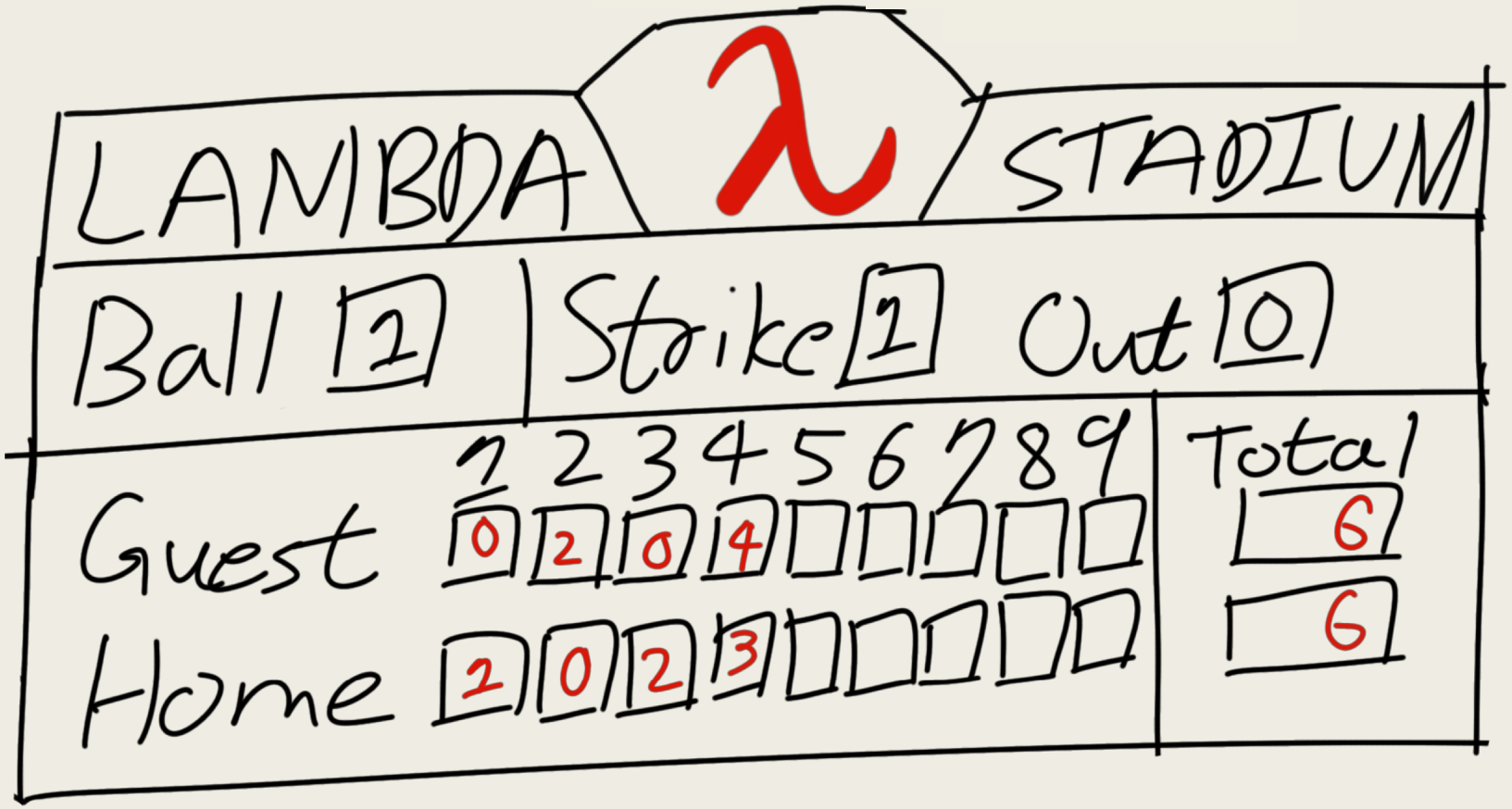
Map Operator

Use when you want to change data

```
function getCities(customers){
   var result = [];
   for(var i = 0; i < customers.length; i++){
      var customer = customers[i];
      result.add(customer.address.city)
   }
   return result;
}

var result = customers.map(x => x.address.city);
```

Reduce (JS)


LAMBDA
λ
STADIUM
Ball 2
Strike 2
Out 0
1 2 3 4 5 6 7 8 9
Total
Guest 0 2 0 4
6
Home 2 0 2 3
6

Reduce Operator

Use when you want a tally of data

```
function calculateSpent(customer) {
let total = 0
   for (let i = 0; i < customer.purchases.length; i++)
   {
      let purchase = customer.purchases[i];
      total = purchase.amount + total;
   }
   return total;
}

let total = customer.purchases
   .reduce((total, x) => total = x.amount + total, 0);
```

All Together Now

Say we didn't customer.amountSpent

And we now only want Customer IDs

```
function CustomerReport(customers, city, amountSpentRequired) {
    let result = [];
    for (let i = 0; i < customers.length; i++) {
        let customer = customers[i];
        if (customer.address.city === city) {
            let amountSpent = 0;
            for (let i = 0; i < customer.purchases.length; i++) {
                let purchase = customer.purchases[i];
                amountSpent = purchase.amount + amountSpent;
            }
            if (amountSpent >= amountSpentRequired)
                result.push(customer.customerId)
        }
    }
    return result;
}
```

All Together Now

Now using Map, Filter, Reduce

```
let resultD = allCustomers
    .filter(x => x.address.city === 'KC')
    .filter(x => x.purchases
        .reduce((total, x) => total = total + x.amount, 0) >= 100)
    .map(x => x.customerId);
```

Instead of imperative code **cluttering the code**, taking a more functional approach puts the **logic we care about front and center**

In Summary
Because we've covered a lot

- Functional Programming is a kind of Declarative Programming – which wants to express what to do by using higher order functions
- Higher order functions either return a function (not covered) or take a function; by taking a function, they can abstract away logic we can ignore
- By using Map, Filter, & Reduce, we can express and focus on the meaningful (business) logic

But What, There's More
Because there's a million more things

- To list a few (far from everything):
  - *Functors, Applicatives, Monads, and all the other Mathematical Theory that Functional Programming is based on*
  - *Data Immutability (plus parallelism benefits)*
  - *Function Composition*
  - *Lazy Evaluation*
  - *A Million Other Things*

Thank You – Questions?

- For the slides & code examples, see:

https://github.com/noah-dev/talk_say_what_not_how

```
// No Fat Arrow Syntax
let resultNFA = allCustomers.filter(function(customer){return customer.address.city === 'KC'})
console.log("Without Fat Arrow Syntax:")
console.log(resultNFA);

// No Fat Arrow Syntax And No Anonymous Function
function checkCity (customer, cityToCheck){
    return customer.address.city === cityToCheck;
}
let resultNFANAF = allCustomers.filter(checkCity, 'KC');
console.log("Without Fat Arrow Syntax And Without Anonymous Function:")
console.log(resultNFA);
```

```
// With Fat Arrow Syntax
let resultWFA = allCustomers.filter(x => x.address.city === 'KC' )
console.log("With Fat Arrow Syntax:")
console.log(resultWFA);

// If logic is more than one line, must use curly braces
// and return statment with fat arrow syntax
let resultMOL = allCustomers.filter(x => {
    let example = false;
    return x.address.city === 'KC' && example ;
})
console.log("Example of more than one line; should return empty")
console.log(resultMOL);
```