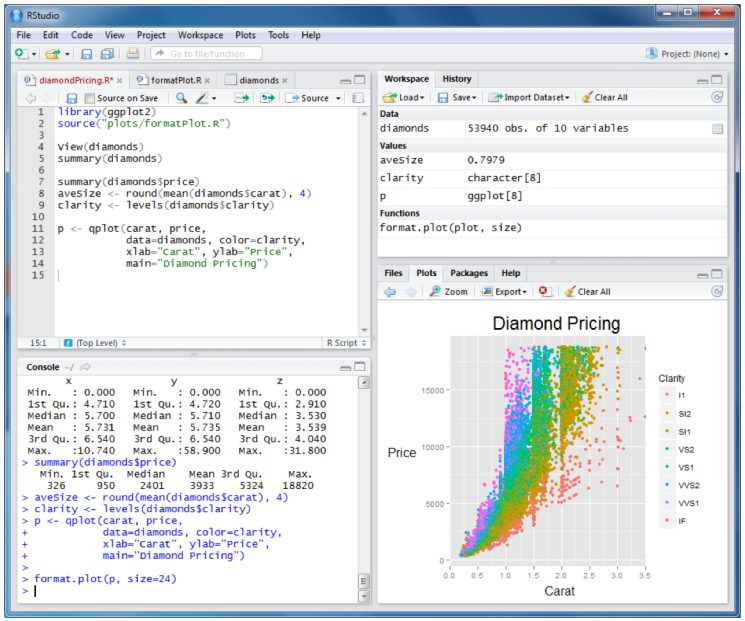
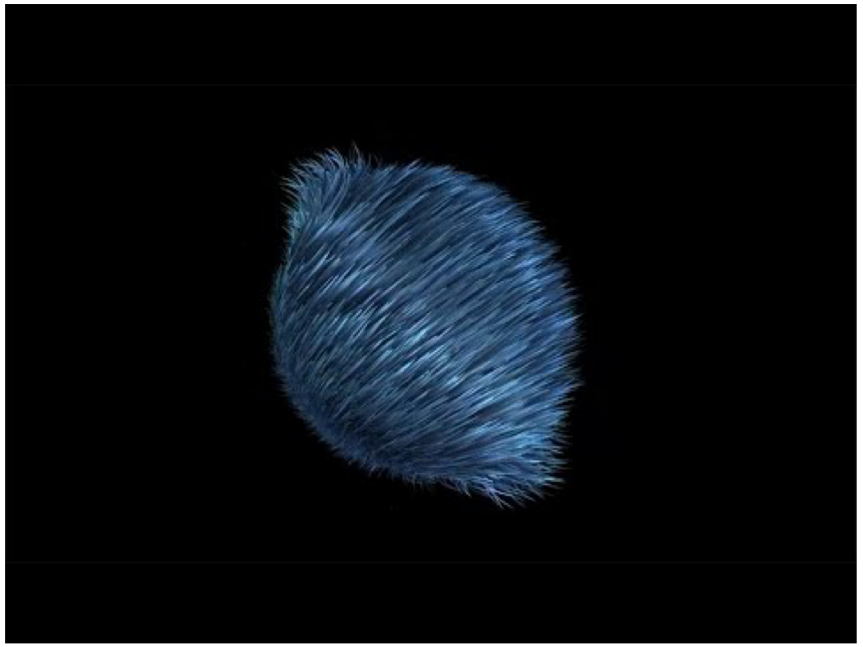
# COMPSCI 220

## Programming Methodology

Slides credit: Arjun Guha

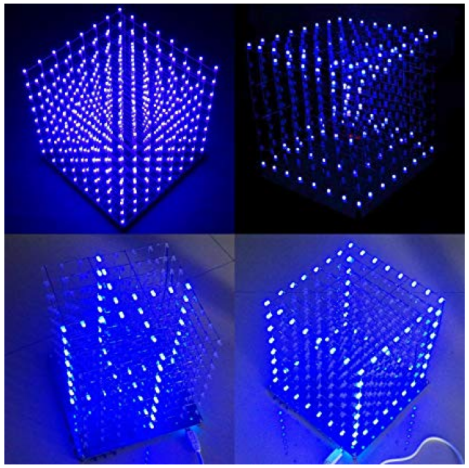
# Different Tasks => Different Languages



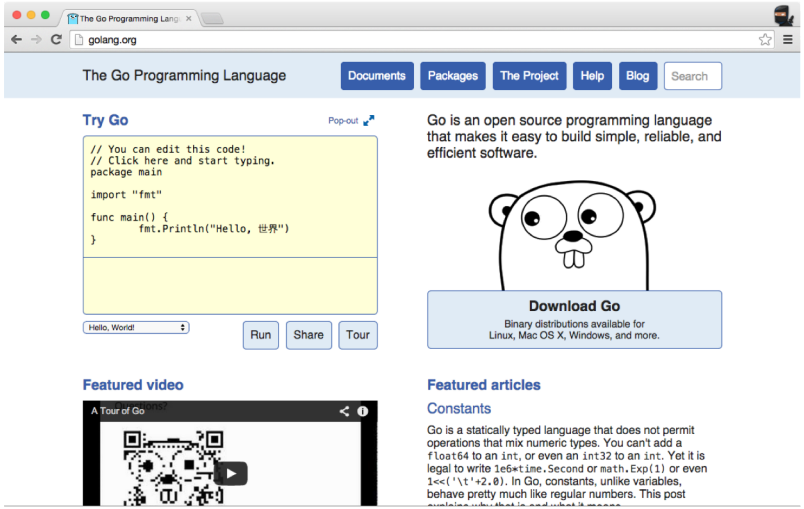



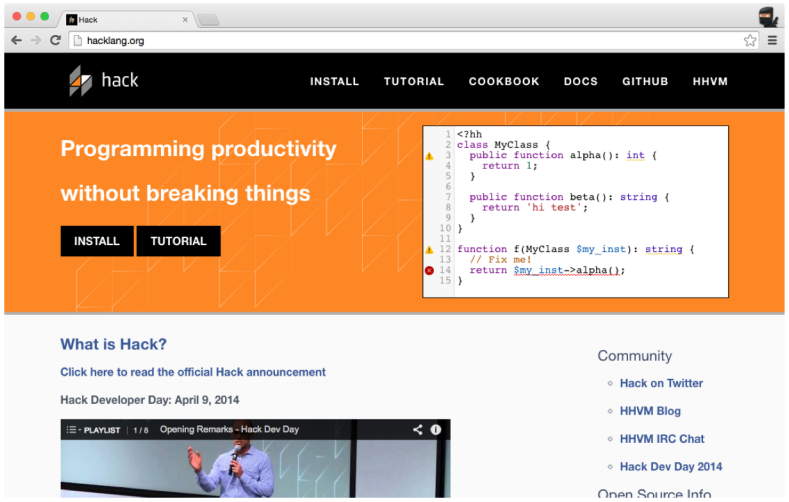
GLShader

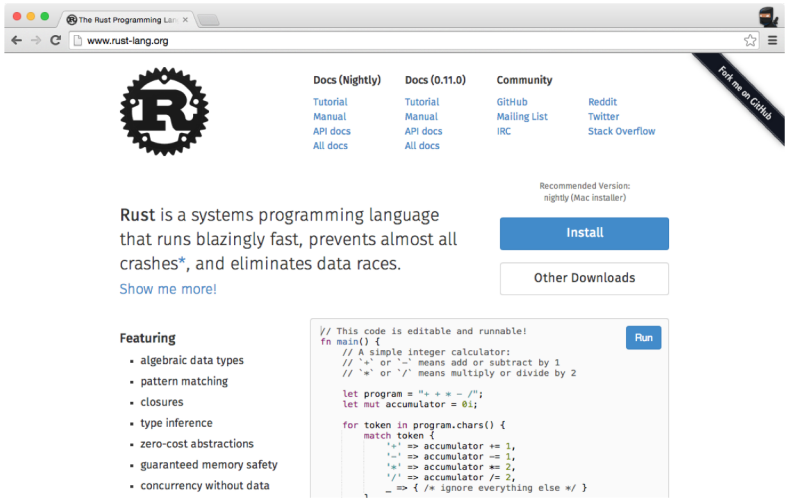
Arduino-C



# New Languages Show Up At An Unprecedented Rate

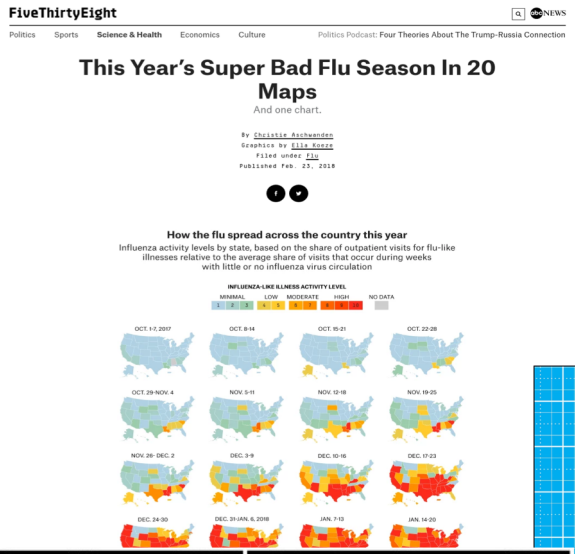
# Our Approach in COMPSCI 220

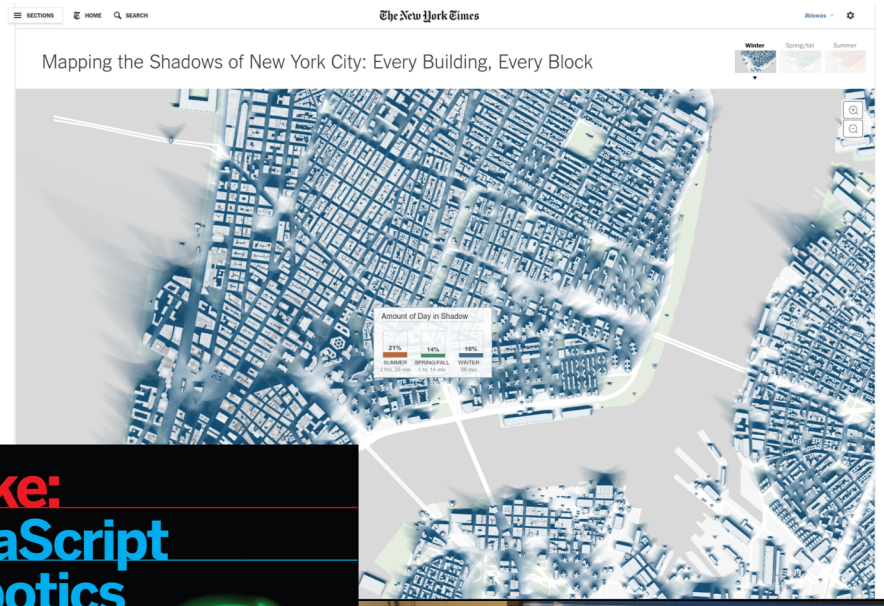
- You will encounter many new languages and will need to master quite a few to succeed in computer science
- We don't have time to teach and learn all of them

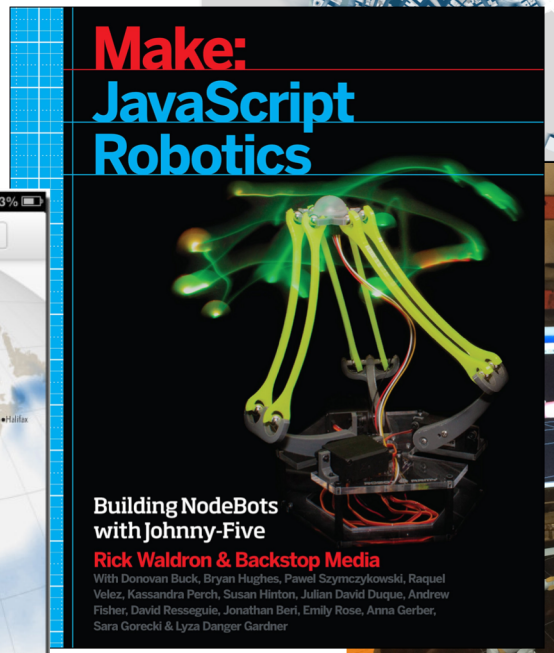
Thus, we will learn:

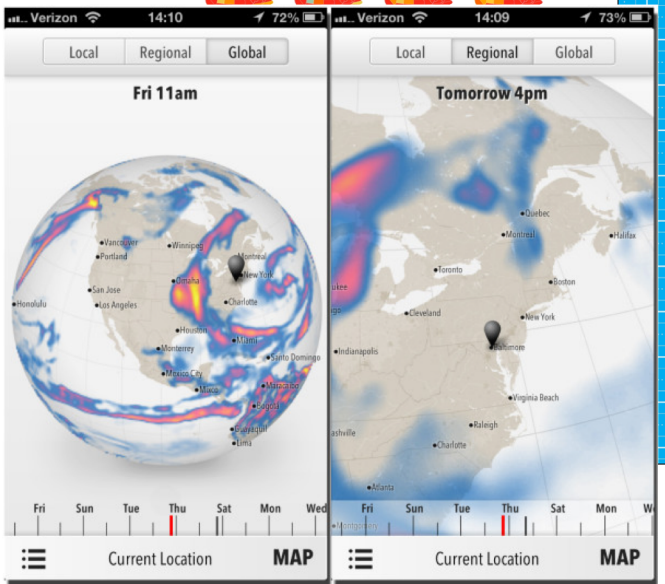
- **Principles** that translate across any modern language, including Java, C++, Scala, Go, Swift
- How to reason about programs for **correctness**, ease of **debugging**, and **maintainability**

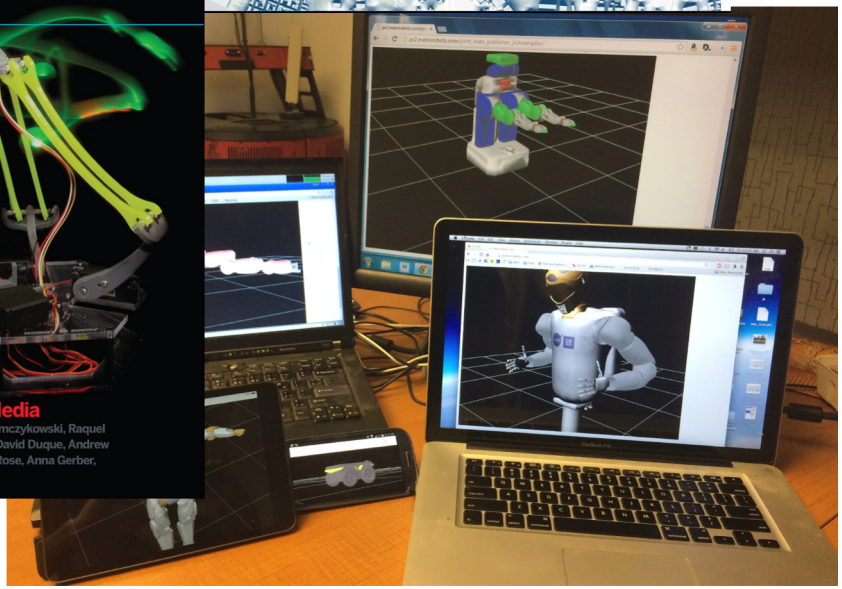
# JavaScript Is Everywhere



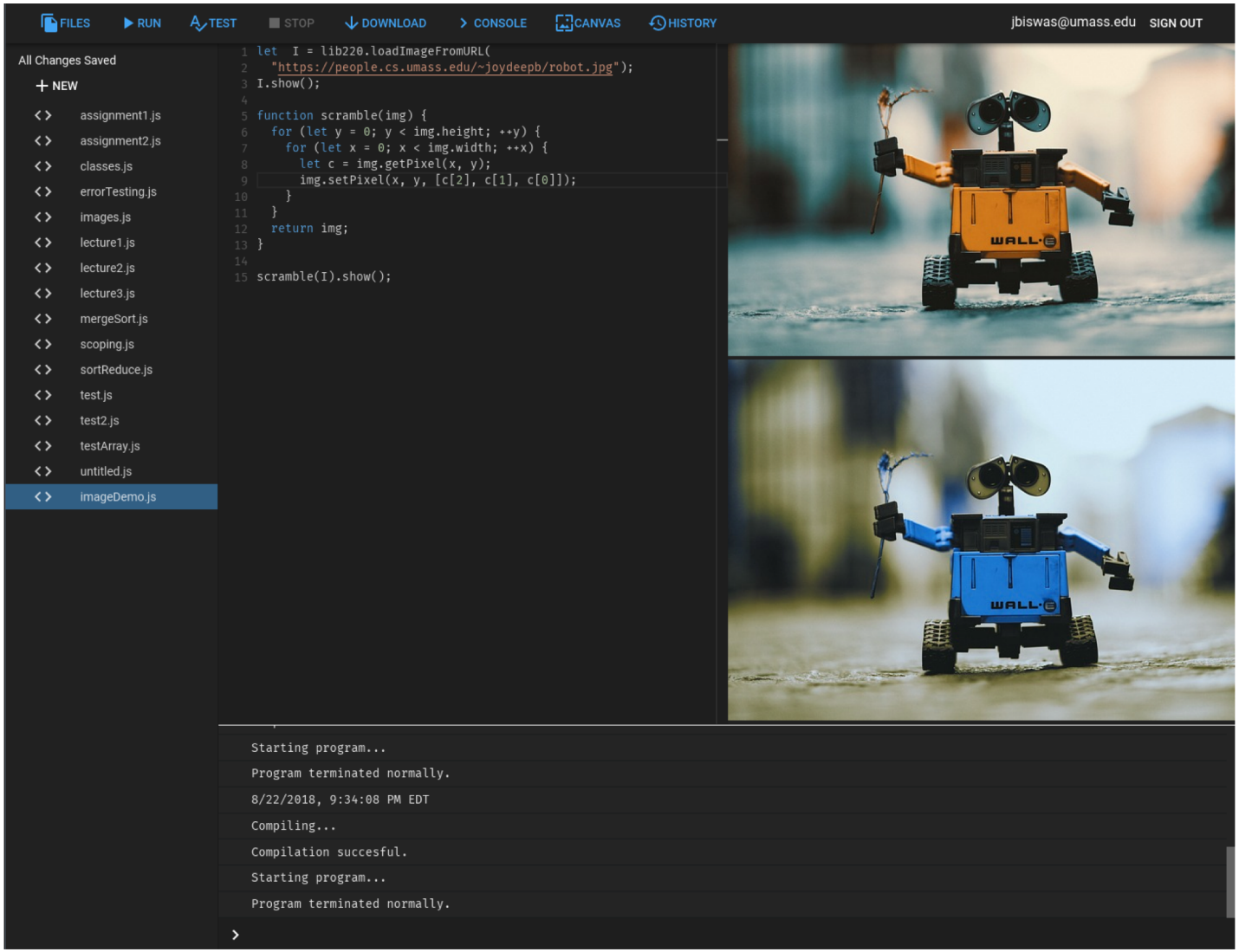



# The 220 web-based IDE: Ocelot



# A Modern Take On JavaScript
# COMPSCI 220-JS: Design to match other languages

1. JavaScript has been around for > 20 years
   - Many old ways of programming still exist
   - New features, abstractions, and capabilities have been continuously added
2. COMPSCI 220: An emphasis on modern JavaScript
   - Checks will disallow older features that have been superseded
   - Known bad coding practices are explicitly prohibited
3. Arbitrary JS code from the internet may not compile
   - Much of the internet uses older JS features, and often poor coding practices
   - You should not be using code from the internet for assignments anyway
4. Anything you write in Ocelot will work outside class

# JS: Basic Syntax; Using the Console

```
let a = 42; // Declare a variable, and assign it a number.
let b = 'Hello World'; // A string.
let c = [0, 1, 4, 9];  // An array of numbers.
const pi = 3.14159265359;  // A constant.
console.log('The value of pi is ' + pi.toString());  // Print to the console.
console.log(c);  // The console can display objects.
let d; // Will not compile: need to initialize value.

// The === operator in JS is equivalent to the == in other languages.
if (1 === 1) {
  console.log('All is good.');
}

function factorial(x) {
  let y = 1;
  for (let z = 1; z <= x; ++z) {
    y = y * z;
  }
  return y;
}
```

# Types in JavaScript

| Java | JavaScript |
| --- | --- |
| `int a = 2;`<br>`float b = 1.3;`<br>`double c = 1.3;`<br>`String d = "Hello world";` | `let a = 2;  // All numbers are doubles`<br>`let b = 1.3;`<br>`let c = 1.3;`<br>`let d = "Hello world";` |

1. *Static types*: In Java, you have to specify the types of variables and methods. The Java compiler catches type errors before you run the program.
2. *Dynamic types*: In JavaScript, you do not have to specify the types in the program. The JavaScript evaluator catches type errors when the program is running.
3. **Note**: JavaScript has types.

```
> typeof 1
"number"
> typeof 2.3
"number"
> typeof "Hello, world!"
"String"
> typeof true
"boolean"
```

Note: all numbers have the type "number", which is equivalent to the type double in Java.

# Some Small *Gotcha* Points About JS

1. Instead of "==", use "===".

   Instead of "!=", use "!=="

   JavaScript does not treat "==" and "!=" the same way as other languages. We will not use this feature: remember, good programming practices.

2. The type of a variable can change

   Do not deliberately use this feature. It hampers readability, and may lead to bugs.

   ```
   let a = 2;
   a = "Hello world";
   ```

3. The value undefined has type "undefined". It is similar to null in Java.

# Introduction to higher-order functions

We introduce several canonical *higher-order functions* over arrays.

- We derive map by abstracting out the differences between two first-order functions
- We generalize the type of map
- We derive filter in a similar way
- As an exercise, we derive map2

# Three different functions

**Problem Statement:** Write a function that consumes an array of numbers and produces an array of numbers. Each element of the output array should be *the double of* the corresponding element in the input array.

**Example:**

```
doubleAll([10, 5, 2])
// [20, 10, 4]
```

**Type Signature:**

```
doubleAll(
  a: number[]): number[]
```

**Problem Statement:** Write a function that consumes an array of numbers and produces an array of numbers. Each element of the output array should be *the reciprocal of* the corresponding element in the input array.

**Example:**

```
recipAll([10, 5, 2])
// [0.1, 0.2, 0.5]
```

**Type Signature:**

```
recipAll(
  a: number[]): number[]
```

**Problem Statement:** Write a function that consumes an array of numbers and produces an array of numbers. Each element of the output array should be *zero if the corresponding element in the input is negative, or the same as the corresponding element if it is non-negative*.

**Examples:**

```
clipAll([-1, 5, -3])
// [0, 5, 0]
```

**Type Signature:**

```
clipAll(
  a: number[]): number[]
```

# Solutions

```
// doubleAll(a: number[]): number[]
function doubleAll(a) {
  let result = [];
  for (let i = 0; i < a.length; ++i) {
    result.push(2 * a[i]);
  }
  return result;
}

// clipAll(a: number[]): number[]
function clipAll(a) {
  let result = [];
  for (let i =  0; i < a.length; ++i) {
    if (a[i] < 0) {
      result.push(0);
    } else {
      result.push(a[i]);
    }
  }
  return result;
}
```

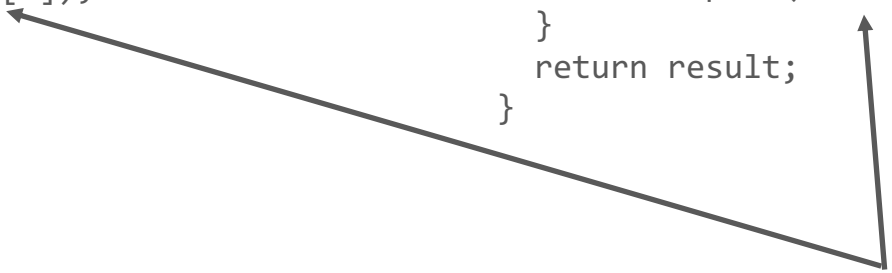
```
// recipAll(a: number[]): number[]
function recipAll(a) {
  let result = [];
  for (let i = 0; i < a.length; ++i) {
    result.push(1 / a[i]);
  }
  return result;
}
```

1. Note: these functions produce a new array and do not update the input array in-place. (See the slides on Unit Testing for why.)
2. These three functions are almost identical. So, they have a lot of duplicated code. Code duplication is bad. How can we address this?

# Step 1: Identify exactly what is different

```
// doubleAll(a: number[]): number[]
function doubleAll(a) {
  let result = [];
  for (let i = 0; i < a.length; ++i) {
    result.push(2 * a[i]);
  }
  return result;
}
```

```
// recipAll(a: number[]): number[]
function recipAll(a) {
  let result = [];
  for (let i = 0; i < a.length; ++i) {
    result.push(1 / a[i]);
  }
  return result;
}
```



Ignore clipAll for a moment.

What is different about about `doubleAll` and `recipAll`?

The only difference between them is what they do to each element of the array.

# Step 2: *Abstract out* the difference

```
// double(x: number): number
function double(x) {
  return 2 * x;
}

// doubleAll(a: number[]): number[]
function doubleAll(a) {
  let result = [];
  for (let i = 0; i < a.length; ++i) {
    result.push(double(a[i]));
  }
  return result;
}
```

```
// recip(x: number): number
function recip(x) {
  return 1 / x;
}

// recipAll(a: number[]): number[]
function recipAll(a) {
  let result = [];
  for (let i = 0; i < a.length; ++i) {
    result.push(recip(a[i]));
  }
  return result;
}
```

Definition: to abstract something out means to write a helper function that performs the operation.

At this point, the only difference between `doubleAll` and `recipAll` is which helper function they apply.

# Step 3: Make the helper function an argument

```
// double(x: number): number
function double(x) {
  return 2 * x;
}
```

```
// recip(x: number): number
function recip(x) {
  return 1 / x;
}
```

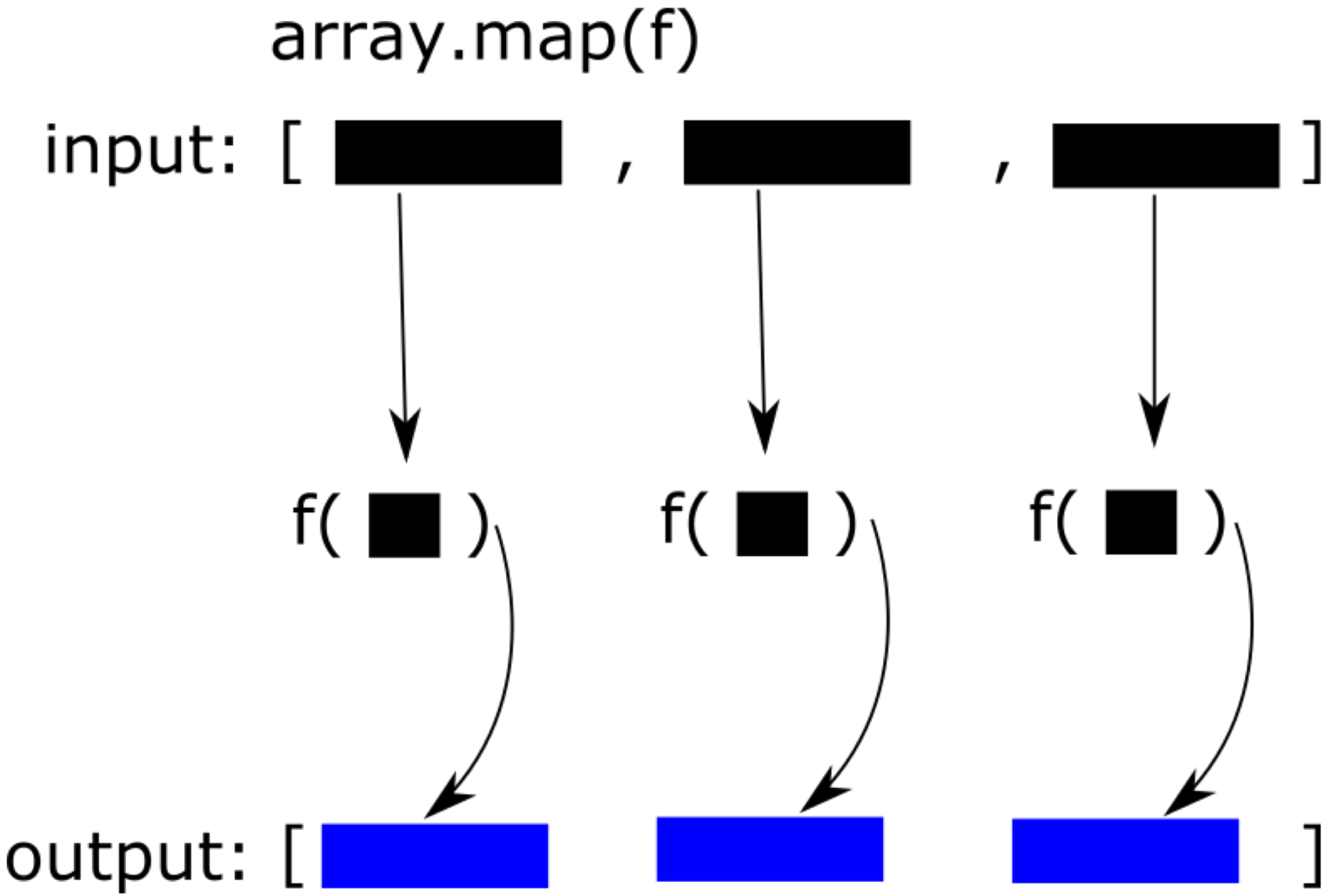
```
// map(f: (x : number) => number, a: number[]): number[]
function map(f, a) {
  let result = [];
  for (let i = 0; i < a.length; ++i) {
    result.push(f(a[i]));
  }
  return result;
}
```

A *higher-order function* is a function that takes another function as an argument.

```
// doubleAll(a: number[]): number[]
function doubleAll(a) {
  return map(double, a);
}
```

```
// recipAll(a: number[]): number[]
function recipAll(a) {
  return map(recip, a);
}
```

# Map, pictorially



# Can we write `clipAll` using `map`?

`map` applies the same function f to each element of the array. `clipAll` appears to do two different things to each element.

```
// clipAll(a: number[]): number[]
function clipAll(a) {
  let result = [];
  for (let i =  0; i < a.length; ++i) {
    if (a[i] < 0) {
      result.push(0);
    } else {
      result.push(a[i]);
    }
  }
  return result;
}
```

```
// clip(x: number): number
function clip(x) {
  if (x < 0) {
    return 0;
  } else {
    return x;
  }
}

// clipAll(a: number[]): number[]
function clipAll(a) {
  let result = [];
  for (let i =  0; i < a.length; ++i) {
    result.push(clip(a[i]));
  }
  return result;
}
```

```
// map(f: (x : number) => number,
       a: number[]): number[]
function map(f, a) {
  let result = [];
  for (let i = 0; i < a.length; ++i) {
    result.push(f(a[i]));
  }
  return result;
}
```

At this point, it is easy to use map. But, it wasn't obvious to start.

# Another example with map

**Problem Statement:** Write a function that consumes an array of string and produces an array of numbers. Each element of the output array should be *the length of the* corresponding string in the input array.

**Example:**

```
lengthAll(["xy", "x"])
// [2, 1]
```

**Type Signature:**

```
lengthAll(
  a: string[]): number[]
```

Can we write this using `map`? We said earlier that the argument `f` to `map` has the type `(x: number) => number`.

```
// length(s: string): number
function length(s) {
  return s.length;
}

// lengthAll(a: string[]): number[]
function lengthAll(a) {
  let result = [];
  for (let i =  0; i < a.length; ++i) {
    result.push(length(a[i]));
  }
  return result;
}
```

```
// map(f: (x : number) => number, a: number[]): number[]
function map(f, a) {
  let result = [];
  for (let i = 0; i < a.length; ++i) {
    result.push(f(a[i]));
  }
  return result;
}
```

# Generalizing the type of map

```
// map<S,T>(f: (x : S) => T, a: S[]): T[]
function map(f, a) {
  let result = [];
  for (let i = 0; i < a.length; ++i) {
    result.push(f(a[i]));
  }
  return result;
}
```

The type of values produced by f are the same as the type of element in the output array.

The type of each element of a must be the same as the type of argument to *f*.

# Three more functions

**Problem Statement:** Write a function that consumes an array of numbers and produces an array of the even numbers in the input array.

**Example:**

```
evens([10, 5, 2])
// [10, 4]
```

**Type Signature:**

```
evens(
  a: number[]): number[]
```

**Problem Statement:** Write a function that consumes an array of numbers and produces an array of the positive numbers in the input array.

**Example:**

```
odds([10, 5, 2])
// [5]
```

**Type Signature:**

```
odds(
  a: number[]): number[]
```

**Problem Statement:** Write a function that consumes an array of strings and produces an array of the non-empty strings in the input array.

**Examples:**

```
nonEmptys(["hi", ""])
// ["hi"]
```

**Type Signature:**

```
nonEmptys(
  a: string[]): string[]
```

# Step 1. Write the functions and identify the differences

```
// evens(a: number[]): number[]
function evens(a) {
  let result = [];
  for (let i = 0; i < a.length; ++i) {
    let x = a[i];
    if (x % 2 === 0) {
      result.push(x);
    }
  }
  return result;
}
```

```
// odds(a: number[]): number[]
function evens(a) {
  let result = [];
  for (let i = 0; i < a.length; ++i) {
    let x = a[i];
    if (x % 2 === 1) {
      result.push(x);
    }
  }
  return result;
}
```

Note: We cannot write these using map, because the length of the output array is not the same as the length of the input array.

# Step 2. Abstract out the differences

```
// isEven(x: number): boolean
function isEven(x) {
  return x % 2 === 0;
}

// evens(a: number[]): number[]
function evens(a) {
  let result = [];
  for (let i = 0; i < a.length; ++i) {
    let x = a[i];
    if (isEven(x)) {
      result.push(x);
    }
  }
  return result;
}
```

```
// isOdd(x: number): boolean
function isOdd(x) {
  return x % 2 === 0;
}

// odds(a: number[]): number[]
function evens(a) {
  let result = [];
  for (let i = 0; i < a.length; ++i) {
    let x = a[i];
    if (isOdd(x)) {
      result.push(x);
    }
  }
  return result;
}
```

At this point, the only difference between `evens` and `odds` is which helper function they apply.

# Step 3: Make the helper function an argument

```
// isEven(x: number): boolean
function isEven(x) {
  return x % 2 === 0;
}
```

```
// isOdd(x: number): boolean
function isOdd(x) {
  return x % 2 === 0;
}
```

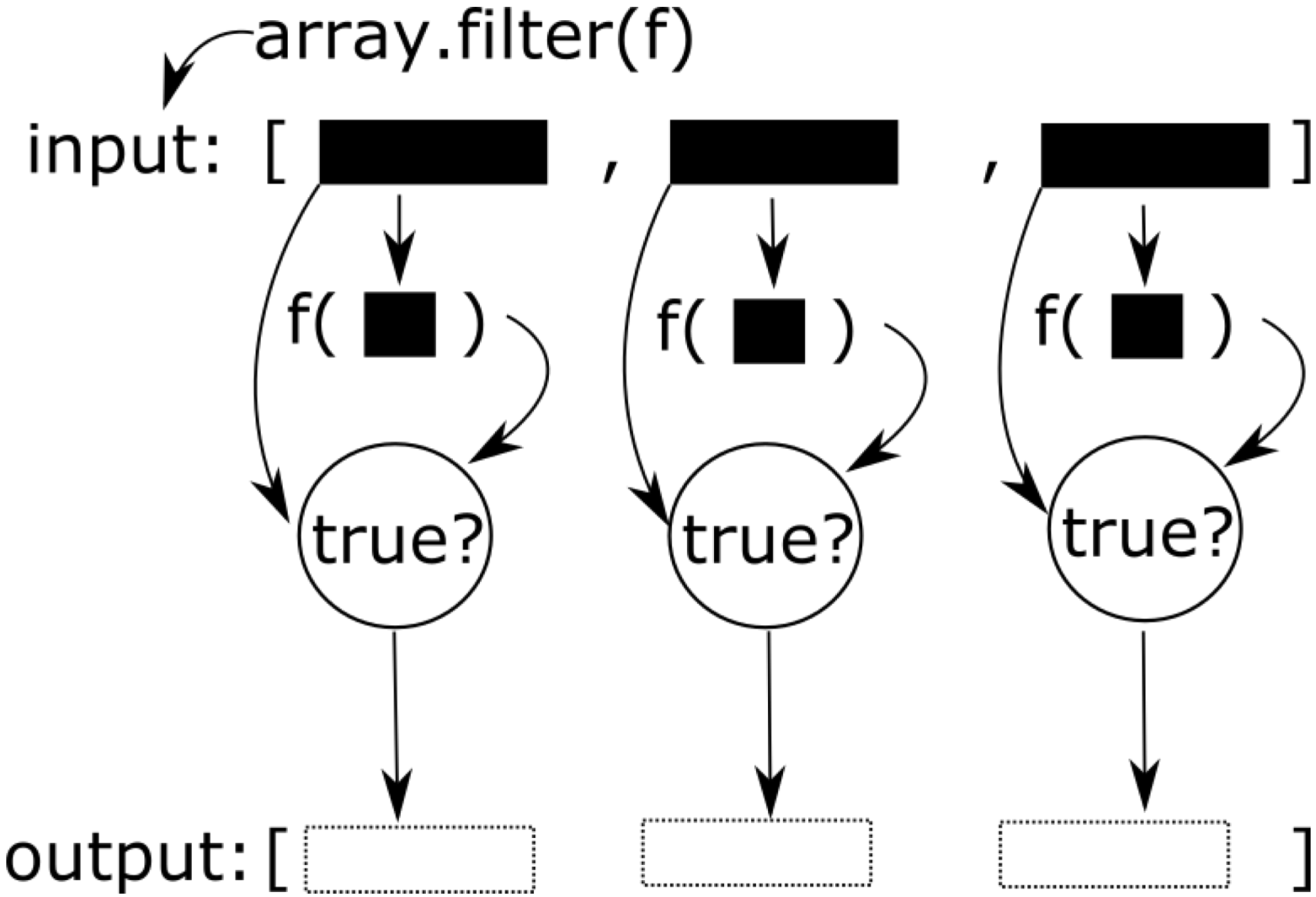
```
// filter<T>(f: (x : T) => boolean, a: T[]): T[]
function filter(f, a) {
  let result = [];
  for (let i = 0; i < a.length; ++i) {
    let x = a[i];
    if (f(x)) {
      result.push(x);
    }
  }
  return result;
}
```

Note that filter does not require the argument to *f* to be a function.

```
// evens(a: number[]): number[]
function evens(a) {
  return filter(isEven, a);
}
```

```
// odds(a: number[]): number[]
function odds(a) {
  return filter(isOdd, a);
}
```

# Filter



# Exercise: two more functions

**Problem Statement:** Write a function that consumes two array of numbers and produces an array of numbers, where each element is the sum of the corresponding numbers in the input array.

**Problem Statement:** Write a function that consumes two array of numbers and produces an array of numbers, where each element is the product of the corresponding numbers in the input array.

Try to derive a higher-order function that you can use to implement both of these functions systematically:

1. Write down some example inputs and outputs for each function.
2. Write the type signature of each function.
3. Implement each function without using any higher-order functions.
4. Abstract out the difference between them into a helper function.
5. Make the helper function an argument.

# Built-in higher-order functions

The JavaScript standard library has several built-in higher-order functions.

# The built-in map function

## Syntax

```
var new_array = arr.map(function callback(currentValue[, index[, array]]) {
    // Return element for new_array
}[, thisArg])
```

### Parameters

**`callback`**
Function that produces an element of the new Array, taking three arguments:

**`currentValue`**
The current element being processed in the array.

**`index`** Optional
The index of the current element being processed in the array.

**`array`** Optional
The array `map` was called upon.

**`thisArg`** Optional
Value to use as `this` when executing `callback`.

### Return value

A new array with each element being the result of the callback function.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

# The built-in map function

let | const

## Syntax

```
var new_array = arr.map(function callback(currentValue[, index[, array]]) {
    // Return element for new_array
}[, thisArg])
```

## Parameters

**`callback`**
Function that produces an element of the new Array, taking three arguments:

**`currentValue`**
The current element being processed in the array.

**`index`** Optional
The index of the current element being pro

**`array`** Optional
The array `map` was called upon.

**`thisArg`** Optional
Value to use as `this` when executing `callba`

## Return value

A new array with each element being the result of the callback function.

**Do not use these. The map function callback should not use these.**

**Recall: Emphasis on good programming practices**

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

# How to Read the Documentation

1. Feel free to try out the many built-in functions
   https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/
2. Do not use 'var'
   Use 'let' or 'const'
   Why? 'var' has unusual scoping rules. Will see after lecture 4
3. Do not use the optional arguments
   - Some are JavaScript's poetic license at re-implementing well-understood standard HOFs (e.g. map, reduce, filter)
   - Some are just bad programming practices.
4. Remember: Check in Ocelot
   !!! Does not run in Ocelot === 0 on assignment

# More Built-In HOFs: forEach(), filter()

```
// Print all entries in array
console.log("Print all entries:");
[1, 2, 3, 4].forEach(
  function(x) {
    console.log(x);
  }
);

// Remember, HOFs take in functions: these can also be built-in functions.
[1, 2, 3, 4].forEach(console.log);

// Filter arrays
console.log("Filter array:");
console.log([1, 2, 3, 4, 5, 6, 7].filter(
  function(x) {
    return (x % 2 === 0);
  }
));
```

# Dealing with errors

Unfortunately, if you program uses higher-order functions has a bug, you will get some inscrutable error messages from JavaScript.

# Common Errors in Calling HOFs

1. Calling a function instead of passing it as a value
```
[1, 2, 3, 4].forEach(console.log(x)); // Incorrect
[1, 2, 3, 4].forEach(console.log); // Correct
```

2. Mixing up parameters passed to HOF, vs. parameters passed to function, passed to HOF
```
[1, 2, 3, 4].forEach(x, function() { console.log(x); });
[1, 2, 3, 4].forEach(function(x) { console.log(x); });
```

3. Return type of function passed to HOF is incorrect
```
[1, 2, 3, 4].filter(function(x) { return '0'; });
[1, 2, 3, 4].filter(function(x) { return false; });
```

4. Function passed to HOF accepts incorrect number of parameters
```
[1, 2, 3, 4].map(function() { return 0; });
[1, 2, 3, 4].map(function(x) { return 0; });
```

5. Function passed to HOF accepts incorrect types of parameters
```
[1, 2, 3, 4].forEach(function(x) { console.log(x[0]); });
[1, 2, 3, 4].forEach(function(x) { console.log(x); });
```

# Dealing With Error Messages: THINK Before you Act!

```
function reciprocal(x) {
  return 1/x;
}
console.log([1, 2, 3, 4].map(reciprocal(x)));
```

```
> x is not defined at Line 4
```

---

```
function reciprocal(x) {
  return 1/x;
}
let x = 0;
console.log([1, 2, 3, 4].map(reciprocal(x)));
```

```
> f is not a function at Line 201: in (anonymous function)
... Line 5
```

# Dealing With Error Messages: THINK Before you Act!

```
function reciprocal(x) {
  return 1/x;
}
function f(x) {
}
let x = 0;
console.log([1, 2, 3, 4].map(reciprocal(x)));
```

```
> f is not a function at Line 201: in (anonymous function)
... Line 7
```

---

```
function apply(x, f) {
  let result = f(x);
  return result;
}
apply(2, 3);
```

```
> f is not a function at Line 2: in apply
... Line 5
```

# Exercise: Fix this error

**Question**: Using map, write a new function called `zeroClone()` to create a new array of the same length as an array provided, but initialized all to zeroes.

```
function zeroClone(a) {
  let a2 = a.map(function() { return 0; });
  return a2;
}
console.log(zeroClone([1, 2, 3, 4]));
```

```
function (anonymous) expected 0 arguments but received 1 argument
at Line 2: in (anonymous function)
... Line 201: in (anonymous function)
... Line 2: in zeroClone
... Line 5
```