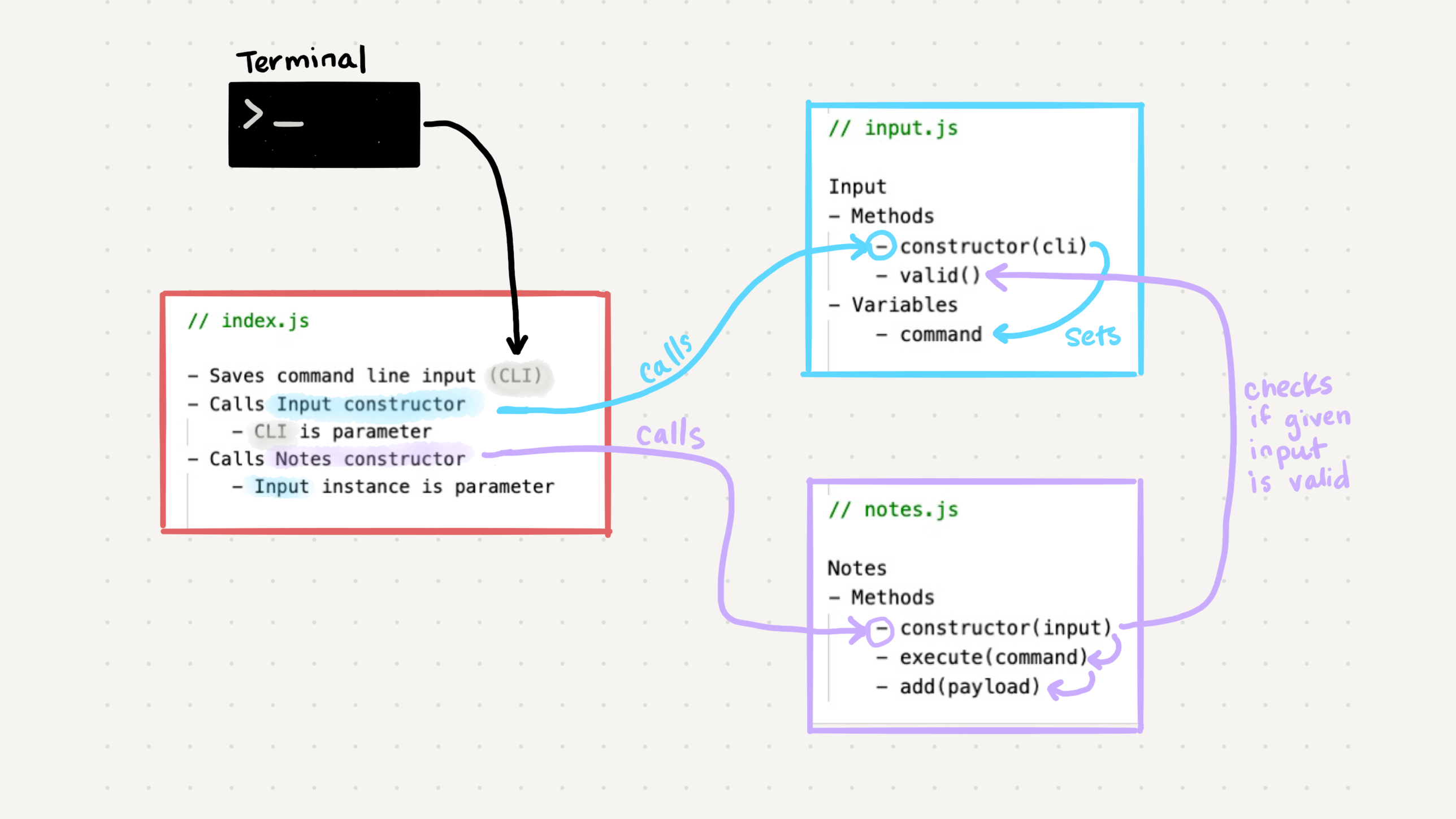
Class 02

Classes, Inheritance & Functional Programming

seattle-javascript-401n16

Lab 01…
what did we do??


Terminal
>_
// index.js
- Saves command line input (CLI)
- Calls Input constructor
- CLI is parameter
- Calls Notes constructor
- Input instance is parameter
Calls
Calls
// input.js
Input
- Methods
- constructor(cli)
- valid()
- Variables
- command
Sets
checks if given input is valid
// notes.js
Notes
- Methods
- constructor(input)
- execute(command)
- add(payload)

Validation

- We can validate objects based on a **schema**
- A schema is an object
  - Each key is a key we expect to see in our valid data
  - Each key's value is an object
    - The object contains information on what the data key's value should look like (what is its type, is it required, etc)

```
const personRules = {
  id: {type: 'string', required: true},
  name: {type: 'string', required: true},
  age: {type: 'number', required: true},
  children: { type: 'array', valueType: 'string' },
};
```

```
const susan = {
  id:'123-45-6789',
  name:'Susan McDeveloperson',
  age: 37,
  children:[],
};
```

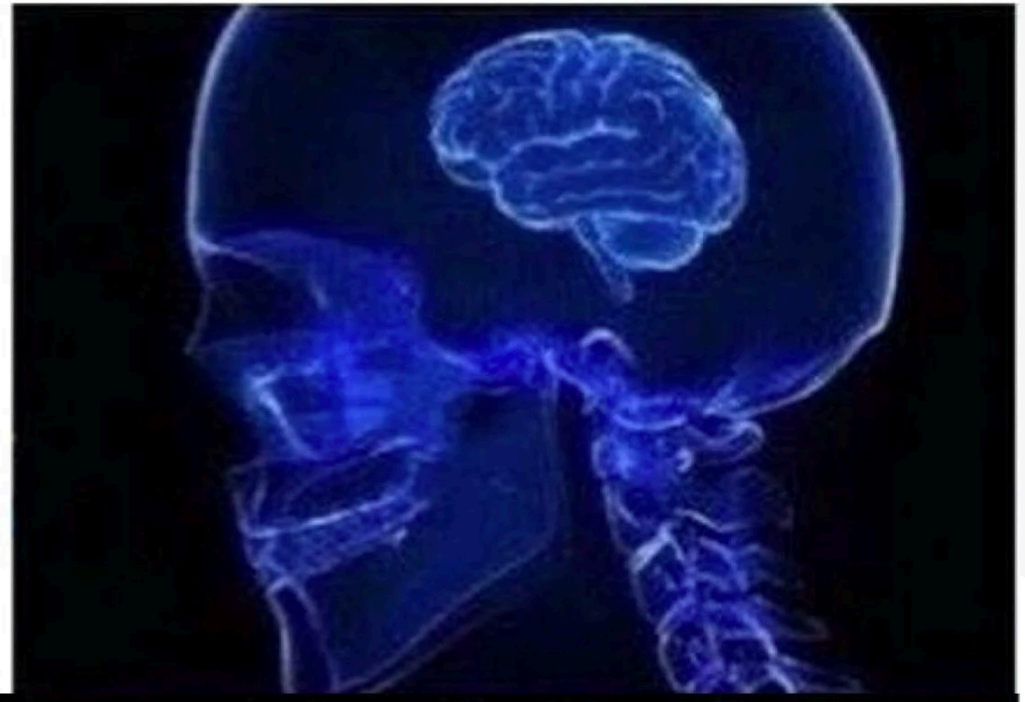
```
const fred = {
  id:38,
  name:'Freddy McCoder',
  children:[],
};
```

Functional Programming

- **What?** A way of thinking about the functions we write
    - **Pure**: A function given the same input should return same output
    - **No Shared State**: One function doesn’t effect how another one runs
    - **Immutable**: Don’t modify the arguments or return value outside of function
- **Why?** We want our functions to be efficient and predictable
- **How?** Break our functions up or use **higher order functions**

Old:

4 lines of code
1 line of comment



Bold:

1 line of code
4 lines of comment

Higher Order Functions

- **Essentially:** A function that takes another function as an argument and/or returns a function

```
[1, 2, 3, 4].map(val => {
    return val * 2;
});
// result: [2, 4, 6, 8]
```

Higher Order Functions

```
newArray = array.map( val => {
  // change val in some way
});

newArray = array.filter( val => {
  // true/false comparison
});

reducedValue = array.reduce( result, val => {
  // use val to change result
});

array.forEach( val, in => {
  // do something with that data
});
```

```
1   'use strict';
2
3   // Functional Programming: Pure Functions
4   // Always returns the same output, given the same input
5   // Causes no side effects
6
7   function multiply(a, b) {
8       return a * b;
9   }
10
11  // Impure (console.log is a side-effect)
12  function multiply(a, b) {
13      console.log(a, b);
14      return a * b;
15  }
16
17  let heightRequirement = 46;
18
19  // Reliable, repeatable input cannot be guaranteed.
20  function canRide(height) {
21      // This makes for an impure function because it relies on a mutable
        outside of itself.
22      return height >= heightRequirement;
23
24      // This would make it a pure function, because it uses an internally
```

Demo

class-02/ demo/ functional-programming

Making functions that are pure, immutable and with no shared state.

Objects

- Boxes for data/information
- Some boxes are generic - take all kinds of data
- Some are weird shapes
- Some are meant for specific content
- Some have a security guard out front

JavaScript Variable Types

- Doesn’t do a lot to limit the data in your variables
- Other languages have custom boxes for different data types
- Functions are also objects!
- JavaScript allows for “generic object” `{ }`

JavaScript

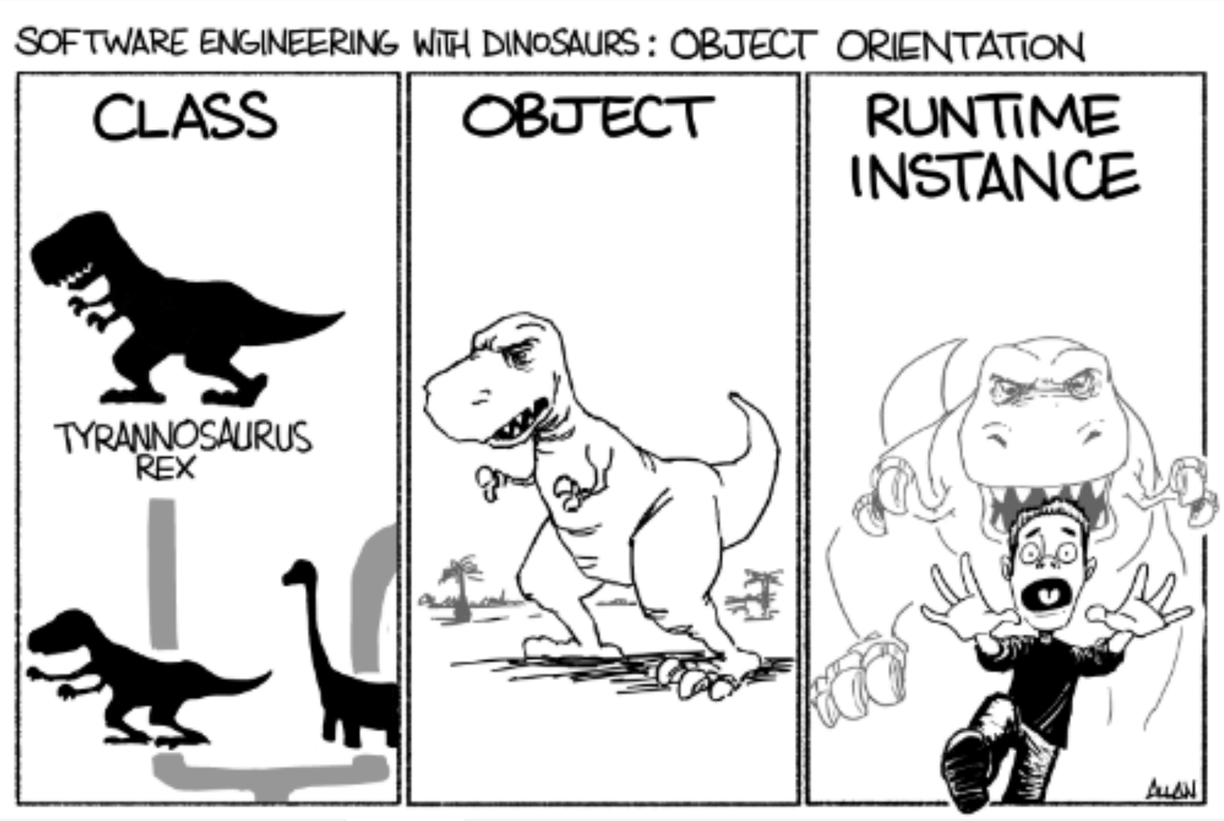
Other Languages

Making Our Own Boxes

- I’m making an application for my pets
- I want to make a custom “box” that can hold the information for my yellow cat, Kitto:
    - Name
    - Age
    - Fur Color
    - Favorite Treat
    - Her jump height

Classes

- We make our own “box types” by defining a **class**
  - A specific classification upon data
  - A blueprint for how what the data should look like
- We then create **instances** of that class
  - The actual created box with contents

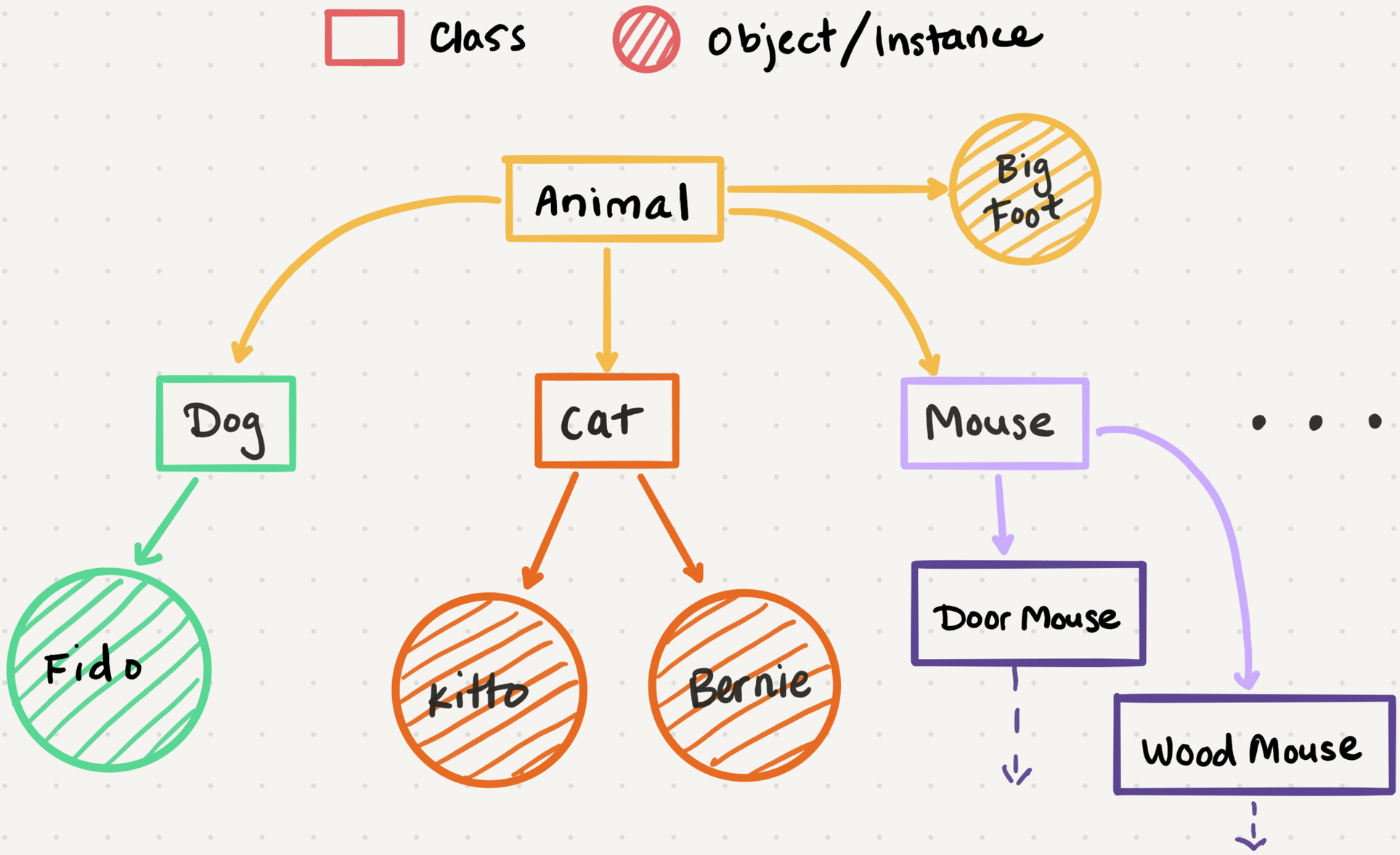


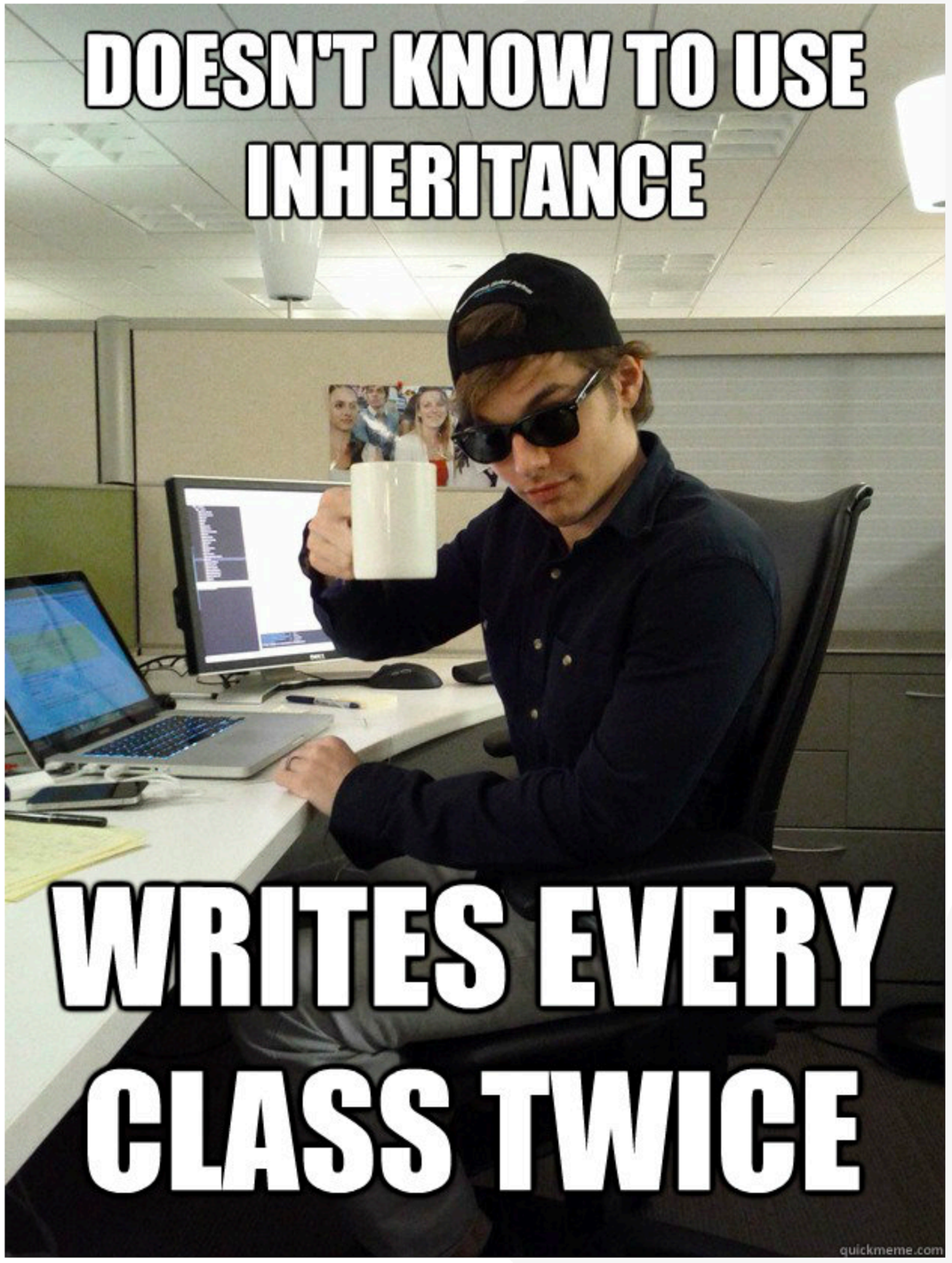
```
'use strict';

// Animal Class
class Animal {
    // static thing = "foobar"; // Node > v 12

    // When creating a new animal, store it's name
    constructor(name) {
        this.name = name;
    }

    // All animals can walk.  This will be a prototype method
    walk() {
        console.log('Walking ... ');
    }
}

// Dogs are animals (extends)
class Dog extends Animal {
    // Only dogs can speak.  This will also be a prototype method.
    speak() {
        console.log('WOOF!');
    }

    // Calling the Animal walk() method when dogs run()
```

Demo

class-01/demo/
classes-inheritance

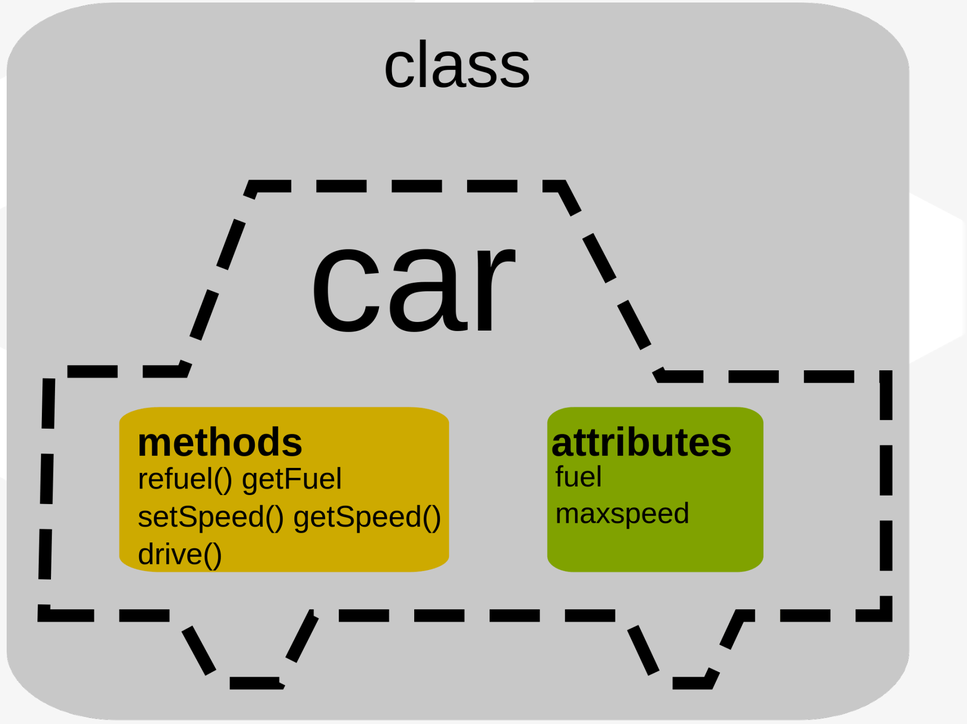
Classes let us define a new data type. **Inheritance** lets us build new classes on top of existing classes. Think of this like an evolutionary tree where each descendent inherits from its ancestors


Class
Object/Instance
Animal
Big Foot
Dog
Cat
Mouse
Fido
kitto
Bernie
Door Mouse
Wood Mouse


DOESN'T KNOW TO USE INHERITANCE
WRITES EVERY CLASS TWICE
quickmeme.com

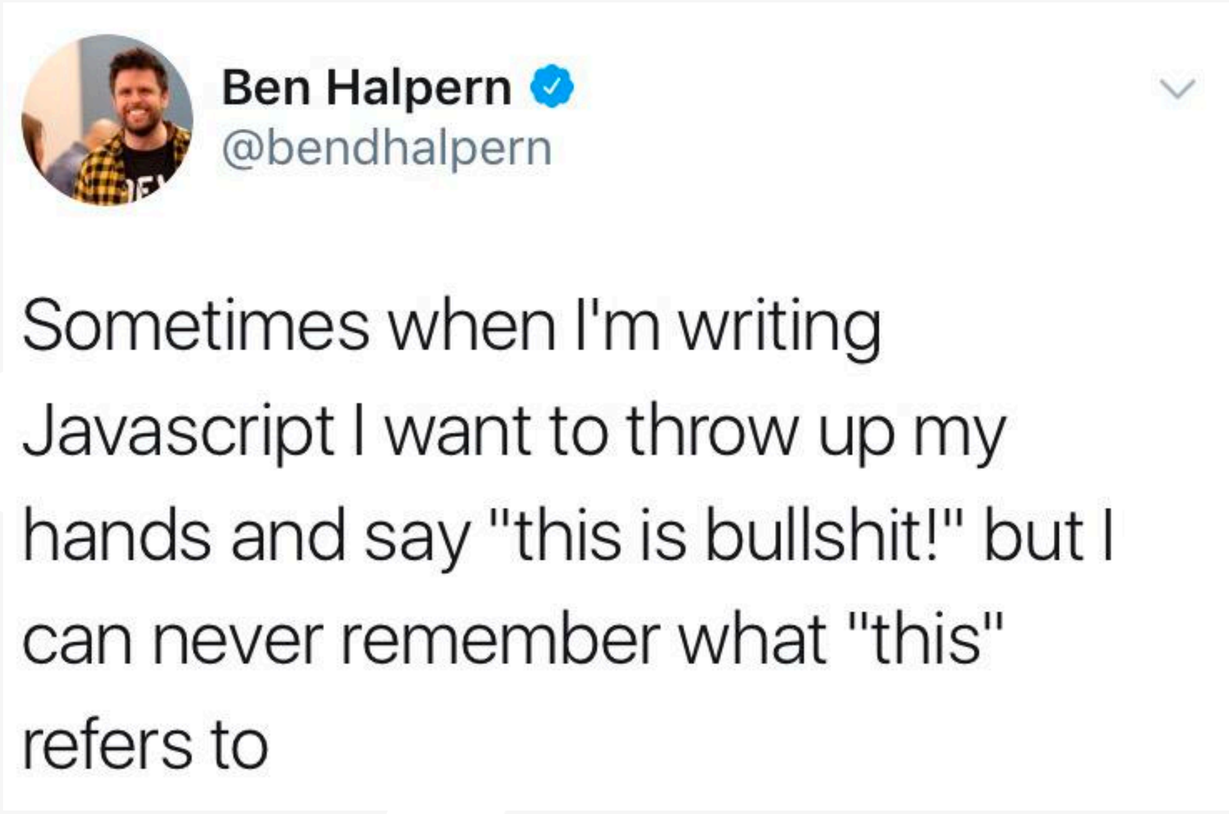
What's In A Class?

- How to **construct** an object of this classification
    - What box is this based on? (`super()`)
    - How to build the box
        - What materials (variables) the box needs to initialize
- What **operations** we can do upon the box
    - Class functions / class methods



Getting To Know `this`

- A handy (but confusing) keyword
- Refers to the **current instance**
- When we're writing a blueprint, we don't have an instance created yet
- We use `this` to specify "the instance once it's created"



Error Handling

- Let’s move away from console logging errors
- Built-in **Error class** we can take advantage of
- Errors are **thrown** - when an error is thrown the program stops
    - Unless it has a **try catch block**!

Demo

class-01/demo/errors

Try Catch Is Here To Save Us

- A `try catch` block will let us respond to a thrown error instead of breaking the app
    - Try to do a thing
    - If an error is thrown, catch it before the app crashes
    - Do something with caught error

Error Types

`Error`	Generic error
`ReferenceError`	An attempt was made to access a variable that is not defined
`SyntaxError`	The JavaScript code was not valid / decipherable
`TypeError`	A provided argument was not the allowable type
`SystemError`	A Node.js error that occurs when a system error has occurred (see `SystemError` table below)

Vocab Review

functional programming

functional programming

A set of guiding principles for software development, where more thought is given to the input and output of functions.

The goal of functional programming is that calling a function anywhere with the same input parameters results in the same output.

pure function

pure function

A pure function is a function where given the same inputs, it always returns the same outputs and has no side effects on the application environment. An example of a side effect is logging to the console.

higher order function

higher order function

A higher order function is any function which takes another function as an argument and/or returns a function. They are often used to abstract or isolate actions within an overarching action.

immutable state

immutable state

Immutable means to not change, and the state of an application usually refers to all of the data an application is maintaining at a certain point in time. Functions which maintain the paradigm of immutable state do not change anything in the application during or after running.

object

object

An object can be a variable, data structure, function or method. Objects, also called *instances*, are a piece of data recorded in memory and accessible by an identifier (such as the variable name or function name).

object-oriented programming (OOP)

object-oriented programming (OOP)

A set of guiding principles where an application is thought of as a collection of objects interacting with one another. The creation of classes helps to solidify complex data collections as a single object.

class

class

A blueprint for an object, specifying how that object should be created, what initial data values it has, and what actions can be done upon this object through class methods. You can think of writing a class as similar to creating a new data type.

prototype

prototype

A prototype is the blueprint of an object.

In JavaScript, everything is based on a generic “Object” class, which is not directly editable by developers. We can change this class blueprint by accessing the prototype of any instance, thereby faking a class specification.

super

`super`

The `super` keyword refers to the parent class of the current class you're developing. By calling `super()`, you essentially call the parent class constructor.

inheritance

inheritance

The process of basing an object or class upon an existing “parent” object or class. This allows the “child” to acquire all the variables and methods specified by the “parent”, making code more DRY.

constructor

constructor

A constructor is a function that defines how an object should be created. Because all JavaScript objects innately inherit from a generic “object” class, we usually don’t have to define much in a constructor other than the initial values of any custom variables.

instance

instance

An instance is a single object created based on a class or data type’s specifications. Instances only exist while the application is running, and in our application code we provide variable name references for them so we can instruct how that instance is used.

context

context

A term used to represent the current scope or environment you're operating in. Any data, variables or functions available to you in that point in time are all within your current context.

this

`this`

A keyword used to refer to the current context; the current object, class, or function you're in. The `this` keyword is usually very helpful in blueprint-type code to refer to itself before any instances of it have been created. Each instance can then interpret `this` to mean themselves.

Your Next Week

Tuesday March 17	Wednesday March 18	Thursday March 19	Friday March 20
6:30 PM **— DUE Class 01 Code Challenge** **— DUE Class 02 Reading** — Class 02A	*6:30 PM* — Class 02B *MIDNIGHT* **— DUE Class 02 Learning Journal**	*6:30 PM* — Class 02 Lab and Code Challenge Co-working *MIDNIGHT* **— DUE Class 01 Lab**	
Saturday March 21	**Sunday March 22**	**Monday March 23**	**Tuesday March 24**
9 AM **— DUE Class 02 Code Challenge** **— DUE Class 02 Lab** **— DUE Class 03 Reading** — Class 03 *MIDNIGHT* **— DUE Class 03 Learning Journal**	*MIDNIGHT* **— DUE Career: Professional Etiquette** **— DUE Class 02 - 03 Feedback**		*6:30 PM* **— DUE Class 03 Code Challenge** **— DUE Class 03 Lab** **— DUE Class 04 Reading** — Class 04A

Lab 02 Overview

Code Challenge 02 Overview

Questions?