
CODESMITH
Software Engineering Residency
Cohort 10 January 2017

Will Sentance

Academic work:
Oxford, Harvard

Currently:
- CEO & Cofounder Codesmith
- Frontend Masters

Previously:
- Cocreator & Engineer @Icecomm
- Software Engineer @Gem

The team that makes it all possible


Will Sentance


Shanda McCune


Schno Mozingo


Eric Kirsten


Haley Godtfredsen


Olivia Leitner


Phil Troutman


Mircea Ilie


Aurora Silva


Christopher Washburn


Jac Chang


Sam Salley


Kaylee Anderson

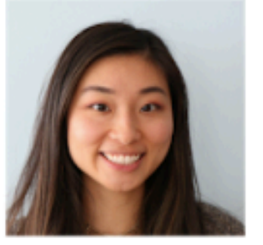

Jenny Mith


Andrew Thomas


Colin McCarthy


Sam Goldberg


William Adamowicz


Stephanie Fong


Joe Thel

I teach software engineering at Codesmith

1. Center of Software Engineering Excellence

- Codesmith student projects have been featured at Google I/O and as Facebook's top developer tool
- Guest mentors include Brian Holt at LinkedIn, Gavin Doughtie at Google, Tom Occhino at Facebook
- Centered in LA and NYC

2. Selective and tight-knit community

- Each selected student has shown enormous potential in five capacities that make an excellent engineer
- Students come from an exceptional and eclectic range of backgrounds from PhDs to software engineers, from Stanford graduates to self-teachers

3. Community of alumni for life

- Graduates receive offers for Mid and Senior Engineer positions (Codesmith graduate salaries range from $95k to $190k)
- Postgraduate education in advanced software architecture and Machine Learning

Recent Codesmith graduates are building at LinkedIn NIKE amazon Microsoft Google

The 5 capacities we look for in candidates

1. Analytical problem solving with code
2. Technical communication (can I implement your approach just from your explanation)
3. Engineering best practices and approach (Debugging, code structure, patience and reference to documentation)
4. Non-technical communication (empathetic and thoughtful communication)
5. Language and computer science experience

Our expectations

- Support each other - engineering empathy is the critical value at Codesmith
- Work hard, Work smart
- Thoughtful communication

Frontend Masters

Javascript the Hard Parts - Object-oriented JavaScript

OOP - an enormously popular paradigm for structuring our complex code

— Easy to add features and functionality
— Easy for us and other developers to reason about (a clear structure)
— Performant (efficient in terms of memory)

We need to organize our code as it gets more complex so it's not just an endless series of commands

Let's suppose we're building a quiz game with users

Some of our users

```
Name: Phil
Score: 4

Name: Julia
Score: 5

Functionality
+ Ability to increase score
```

What would be the best way to store this data and functionality?

Objects - store functions with their associated data!

```
const user1 = {
  name: "Phil",
  score: 4,
  increment: function() {
    user1.score++;
  }
};

user1.increment(); //user1.score => 4
```

This is the principle of encapsulation.

Let's keep creating our objects

Note we would in reality have a lot of different relevant functionality for our user objects

- Ability to increase score
- Ability to decrease score
- Delete user
- Log in user
- Log out user
- Add avatar
- get user score
- … (100s more applicable functions)

What alternative techniques do we have for creating objects?

Creating user2 user 'dot notation'

```
const user2 = {}; //create an empty object

user2.name = "Julia"; //assign properties to that object
user2.score = 5;
user2.increment = function() {
  user2.score++;
};
```

Creating user3 using Object.create

```
const user3 = Object.create(null);

user3.name = "Eva";
user3.score = 9;
user3.increment = function() {
  user3.score++;
};
```

Our code is getting repetitive, we're breaking our DRY principle

And suppose we have millions of users!

What could we do?

Solution 1. Generate objects using a function

```
function userCreator(name, score) {
  const newUser = {};
  newUser.name = name;
  newUser.score = score;
  newUser.increment = function() {
    newUser.score++;
  };
  return newUser;
};

const user1 = userCreator("Phil", 4);
const user2 = userCreator("Julia", 5);
user1.increment()
```

Problems:

Each time we create a new user we make space in our computer's memory for all our data and functions. But our functions are just copies

Is there a better way?

Benefits:

It's simple and easy to reason about!

Solution 2:

Store the `increment` function in just one object and have the interpreter, if it doesn't find the function on `user1`, look up to that object to check if it's there

How to make this link?

Using the prototype chain

```
const functionStore = {
  increment: function(){this.score++;},
  login: function(){console.log("You're loggedin")}
};

const user1 = {
  name: "Phil",
  score: 4
}

user1.name // name is a property of user1 object
user1.increment // Error! increment is not!
```

Link user1 and functionStore so the interpreter, on not finding .increment, makes sure to check up in functionStore where it would find it

Make the link with `Object.create()` technique

```
const user1 = Object.create(functionStore)
user1 // {}

user1.increment // function....
```

Interpreter doesn't find `.increment` on user1 and looks up the prototype chain to the next object and finds `.increment` 1 level up

Solution 2 in full

```
function userCreator (name, score) {
  const newUser = Object.create(userFunctionStore);
  newUser.name = name;
  newUser.score = score;
  return newUser;
};


const userFunctionStore = {
  increment: function(){this.score++;},
  login: function(){console.log("You're loggedin");}
};

const user1 = userCreator("Phil", 4);
const user2 = userCreator("Julia", 5);
user1.increment();
```

Problem

No problems! It's beautiful

Maybe a little long-winded

```
const newUser = Object.create(functionStore);
...
return newUser
```

Write this every single time - but it's 6 words!

Super sophisticated but not standard

Pair Programming

Answer these:

— I know what a variable is
— I've created a function before
— I've added a CSS style before
— I have implemented a sort algorithm (bubble, merge etc)
— I can add a method to an object’s prototype
— I understand the event loop in JavaScript
— I understand 'callback functions'
— I’ve implemented `filter` from scratch
— I can handle collisions in hash tables

Object Oriented JavaScript Pair Programming Challenges

csx.codesmith.io

— Unit 6: Object Oriented Programming
— Gives you tests to check your answers

csbin.io/OOP

— No login needed

Solution 3 - Introducing the keyword that automates the hard work: new

```
const user1 = new userCreator("Phil", 4)
```

When we call the constructor function with new in front we automate 2 things

1. Create a new user object
2. return the new user object

But now we need to adjust how we write the body of userCreator - how can we:

— Refer to the auto-created object?
— Know where to put our single copies of functions?

The new keyword automates a lot of our manual work

```
function userCreator(name, score) {
 const newUser = Object.create(functionStore);
 newUser this.name = name;
 newUser this.score = score;
 return newUser;
};

functionStore userCreator.prototype // {};
functionStore userCreator.prototype.increment = function(){
 this.score++;
}

let user1 = new userCreator("Phil", 4);
```

Interlude - functions are both objects and functions :/

```
function multiplyBy2(num){
    return num*2
}

multiplyBy2.stored = 5
multiplyBy2(3) // 6

multiplyBy2.stored // 5
multiplyBy2.prototype // {}
```

We could use the fact that all functions have a default property on their object version, 'prototype', which is itself an object - to replace our functionStore object

Complete Solution 3

```
function UserCreator(name, score){
  this.name = name;
  this.score = score;
}

UserCreator.prototype.increment = function(){
  this.score++;
};
UserCreator.prototype.login = function(){
  console.log("login");
};

const user1 = new UserCreator(“Eva”, 9)

user1.increment()
```

Benefits

— Faster to write

— Still typical practice in professional code

— 99% of developers have no idea how it works and therefore fail interviews

— We have to upper case first letter of the function so we know it requires ‘new’ to work!

What if we want to organize our code inside one of our shared functions - perhaps by defining a new inner function

```
function UserCreator(name, score){
  this.name = name;
  this.score = score;
}

UserCreator.prototype.increment = function(){
  function add1(){
    this.score++;
  }
  // const add1 = function(){this.score++;}
  add1()
};

UserCreator.prototype.login = function(){
  console.log("login");
};

const user1 = new UserCreator(“Eva”, 9)

user1.increment()
```

We need to introduce arrow functions - which bind `this` lexically

```
function UserCreator(name, score){
  this.name = name;
  this.score = score;
}

UserCreator.prototype.increment = function(){
  const add1 = ()=>{this.score++}
  add1()
};

UserCreator.prototype.login = function(){
  console.log("login");
};

const user1 = new UserCreator(“Eva”, 9)

user1.increment()
```

Solution 4

We're writing our shared methods separately from our object 'constructor' itself (off in the `User.prototype` object)

Other languages let us do this all in one place. ES2015 lets us do so too

The class ‘syntactic sugar’

```
class UserCreator {
  constructor (name, score){
      this.name = name;
      this.score = score;
  }
  increment (){
    this.score++;
  }
  login (){
    console.log("login");
  }
}

const user1 = new UserCreator("Eva", 9);

user1.increment();
```

The Class ‘Syntactic Sugar’

Solution 3 in full

```
function UserCreator(name, score){
  this.name = name;
  this.score = score;
}

UserCreator.prototype.increment = function(){
  this.score++;
};
UserCreator.prototype.login = function(){
  console.log("login");
};

const user1 = new UserCreator(“Eva”, 9)

user1.increment()
```

Solution 4 in full

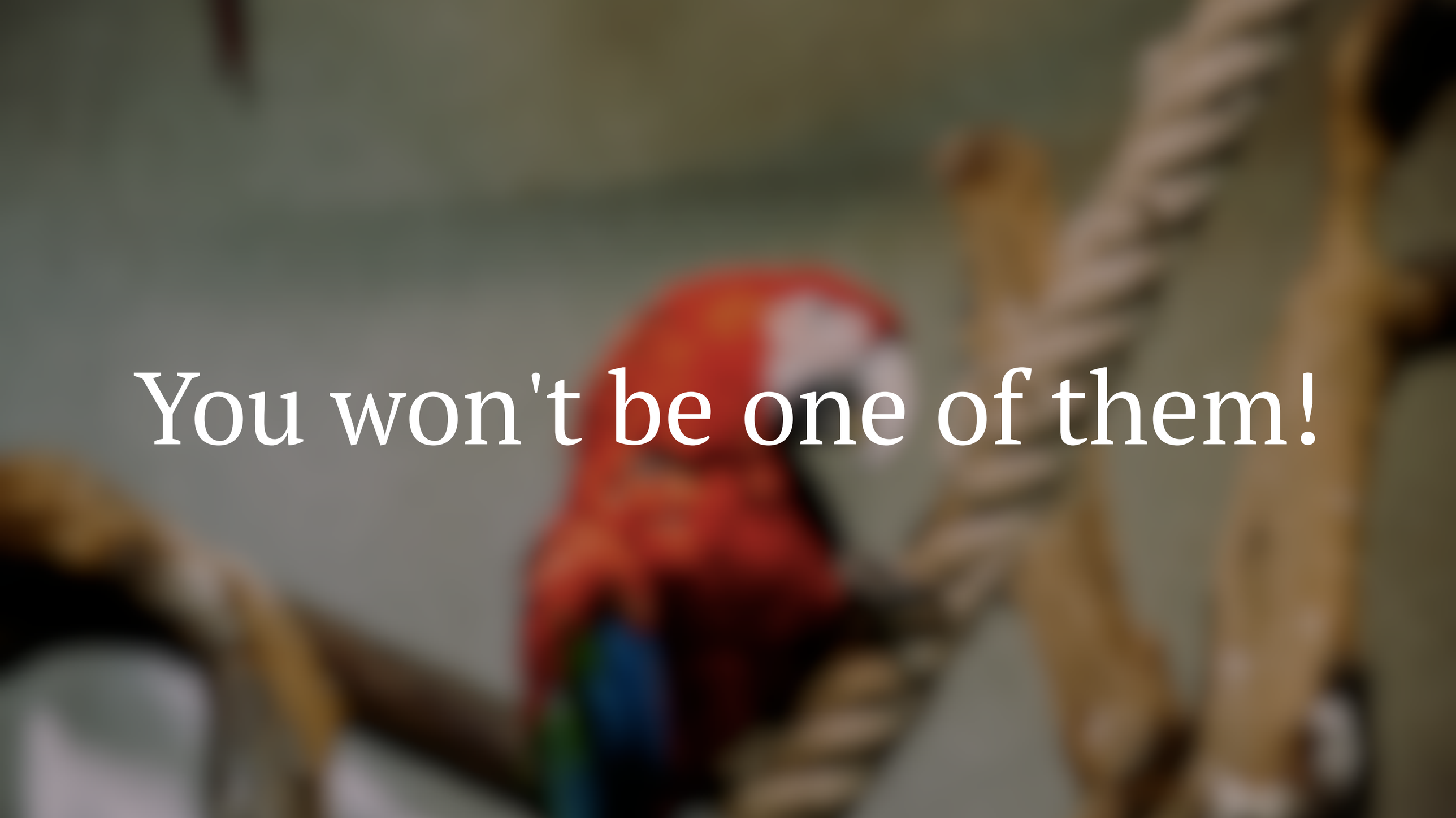
```
class UserCreator {
  constructor (name, score){
      this.name = name;
      this.score = score;
  }
  increment (){
    this.score++;
  }
  login (){
    console.log("login");
  }
}

const user1 = new UserCreator("Eva", 9);

user1.increment();
```

Benefits:

— Emerging as a new standard
— Feels more like style of other languages (e.g. Python)

Problems

— 99% of developers have no idea how it works and therefore fail interviews


You won't be one of them!

JavaScript uses this proto link to give objects, functions and arrays a bunch of bonus functionality. All objects by default have `__proto__`

```
const obj = {
    num : 3
}

obj.num // 3
obj.hasOwnProperty("num") // ? Where's this method?

Object.prototype // {hasOwnProperty: FUNCTION}
```

- With `Object.create` we override the default `__proto__` reference to `Object.prototype` and replace with `functionStore`
- But `functionStore` is an object so *it* has a `__proto__` reference to `Object.prototype` - we just intercede in the chain

Arrays and functions are also objects so they get access to all the functions in Object.prototype but also more goodies

```
function multiplyBy2(num){
  return num*2
}

multiplyBy2.toString() //Where is this method?

Function.prototype // {toString : FUNCTION, call : FUNCTION, bind : FUNCTION}

multiplyBy2.hasOwnProperty("score") // Where's this function?

Function.prototype.__proto__ // Object.prototype  {hasOwnProperty: FUNCTION}
```

Object Oriented JavaScript Pair Programming Challenges

csx.codesmith.io

— Unit 6: Object Oriented Programming
— Gives you tests to check your answers

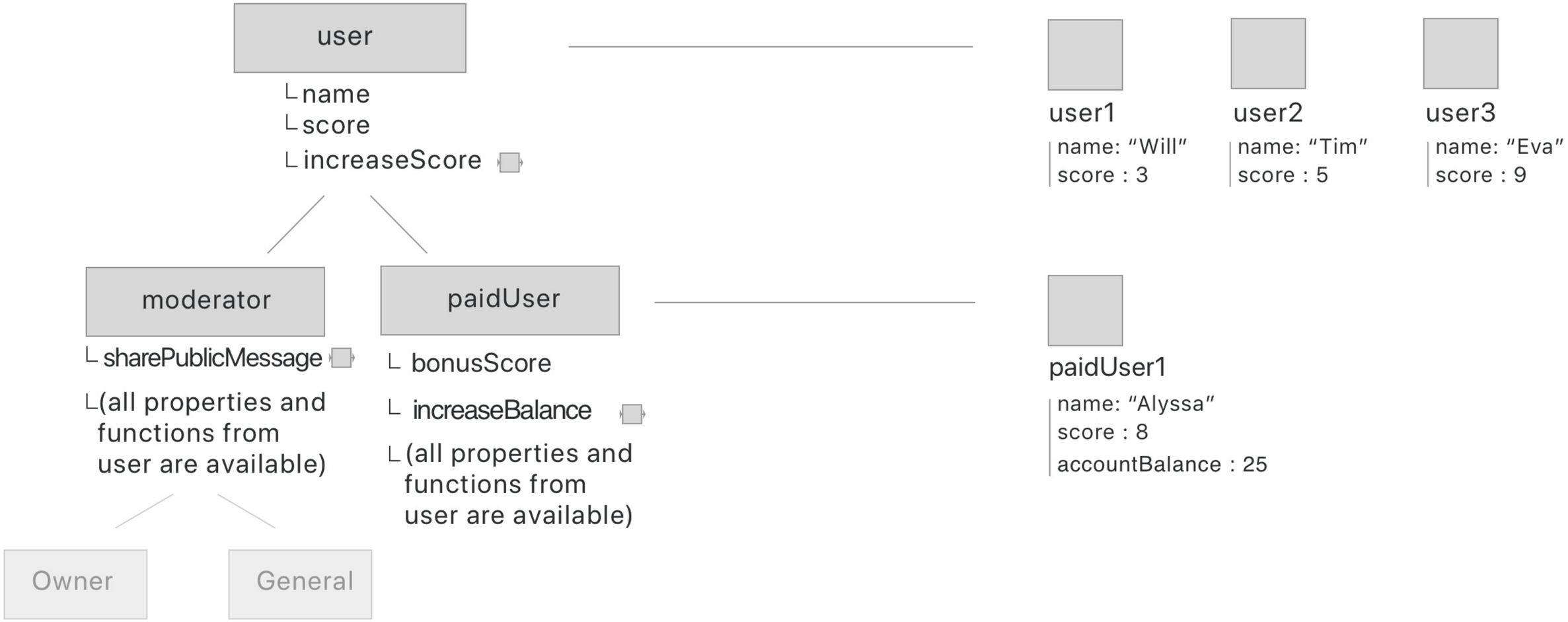
csbin.io/OOP

— No login needed

Subclassing

A core aspect of an OOP approach is inheritance - passing knowledge down

Note the bond between user, moderator and upgradedUser



We can achieve this in JavaScript in Solution 2, 3 and 4

Subclassing in Solution 2

Factory function approach

```
function userCreator(name, score){
  const newUser = Object.create(userFunctions);
  newUser.name = name;
  newUser.score = score;
  return newUser;
}

userFunctions = {
  sayName: function (){
      console.log("I'm " + this.name);
  },
  increment: function(){
      this.score++;
  }
}

const user1 = userCreator("Phil",5);

user1.sayName(); // "I am Phil"
```

```
function paidUserCreator(paidName, paidScore, accountBalance){
  const newPaidUser = userCreator(paidName, paidScore);
  Object.setPrototypeOf(newPaidUser, paidUserFunctions);
  newPaidUser.accountBalance = accountBalance;
  return newPaidUser;
}

const paidUserFunctions = {
  increaseBalance : function (){
    this.accountBalance++;
  }
};

Object.setPrototypeOf(paidUserFunctions, userFunctions)

const paidUser1 = paidUserCreator("Alyssa", 8, 25);

paidUser1.increaseBalance();

paidUser1.sayName(); // "I'm Alyssa"
```

Interlude - We have another way of running a function that allow us to control the assignment of `this`

```
const obj = {
  num: 3,
  increment: function(){this.num++;}
};

const otherObj = {
  num: 10
};

obj.increment(); // obj.num now 4

obj.increment.call(otherObj); // otherObj.num now 11
// obj.increment.apply(otherObj);
```

`this` always refers to the object to the left of the dot on which the function (method) is being called - unless we override that by running the function using `.call()` or `.apply()` which lets us set the value of `this` inside of the `increment` function

Subclassing in Solution 3

Constructor (Pseudoclassical) approach

```
function userCreator (name, score){
  this.name = name
  this.score = score
}
```

```
userCreator.prototype.sayName = function (){
  console.log("I'm " + this.name);
}
userCreator.prototype.increment = function(){
  this.score++;
}
```

```
const user1 = new userCreator("Phil", 5);
const user2 = new userCreator("Tim", 4);

user1.sayName(); // "I'm Phil"
```

```
function paidUserCreator (paidName, paidScore, accountBalance){
  userCreator.call(this, paidName, paidScore);
  // userCreator.apply(this, [paidName, paidScore])
  this.accountBalance = accountBalance;
}

paidUserCreator.prototype = Object.create(userCreator.prototype);

paidUserCreator.prototype.increaseBalance = function (){
  this.accountBalance++;
};

const paidUser1 = new paidUserCreator("Alyssa", 8, 25);

paidUser1.increaseBalance()

paidUser1.sayName() // "I'm Alyssa"
```

Subclassing in Solution 4

ES2015 Class approach

```
class userCreator{
  constructor  (name, score){
    this.name = name;
    this.score = score;
  }
  sayName (){
    console.log("I am " + this.name);
  }
  increment (){
    this.score++;
  }
}

const user1 = new userCreator("Phil", 4);
const user2 = new userCreator("Tim", 4);

user1.sayName()
```

```
class paidUserCreator extends userCreator {
  constructor(paidName, paidScore, accountBalance){
    super (paidName, paidScore);
    this.accountBalance = accountBalance;
  }
  increaseBalance (){
    this.accountBalance++;
  }
}

const paidUser1 = new paidUserCreator("Alyssa", 8, 25);

paidUser1.increaseBalance();

paidUser1.sayName();
```

The Hard Parts Challenge Code

— We created the Hard Parts Challenge code to guarantee an interview for the Hard Parts community members

— We invite ~35% of regular online applications to interview. Completion of the Hard Parts challenge code *guarantees* interview for Codesmith

— It builds upon the content you worked on today

How to continue your JavaScript journey

JS functions, objects, arrays, loops

- Workshop: JavaScript Easier Parts
- CSX Unit 1: Functions and Execution Context

Web Development: HTML, CSS & jQuery

- (CSX Project) Building a Chrome extension

JS Callbacks & Higher order functions

- Workshop: JS the Hard Parts: Callbacks
- CSX Unit 2: Callbacks & Higher order functions

Traditional Coding School

+

JS Closures

- Workshop: JS the Hard Parts: Closures
- CSX Unit 3: Closures

Algorithms & recursion

- Workshop: Recursion
- CSX Unit 4: Recursion

Technical communication

- All JSHP workshops
- (Online) Office hours and pair-programming

Codesmith

- In-person & online workshops
- Online activities
- CSX (codesmith.io/learn)
- Self-guided projects

OOP in JS: Workshop & CSX

Asynchronous JS: Workshop & CSX