Unit 1 - JS Fundamentals

...

OOP in Javascript

Classes, Prototypes & Object-oriented programming

An enormously popular paradigm for structuring our complex code

Easy to add features and functionality

Performant (efficient in terms of memory)

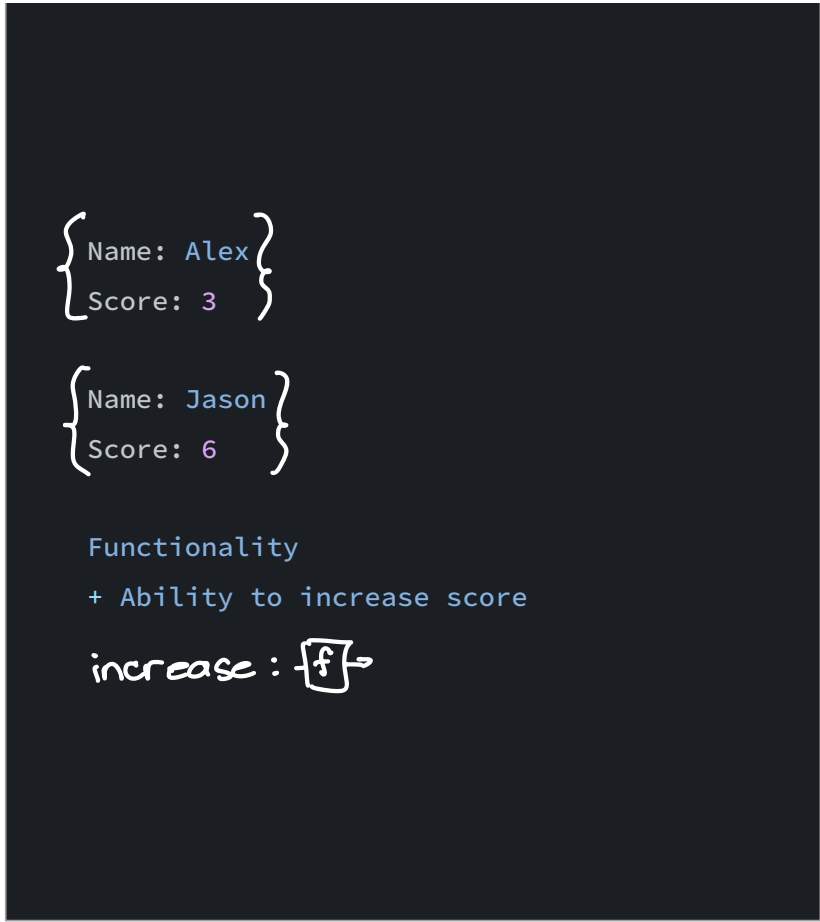
Easy for us and other developers to reason about (a clear structure)

We're building a quiz game with users

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

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)



Objects - store functions with their associated data!

This is the principle of encapsulation.

Let's keep creating our objects

What alternative techniques do we have for creating objects?

```
const user1 = {
 name: "Alex",
 score: 3, 4
 increment: function() {
   user1.score++;
 }
};


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

Creating user2 user *dot notation*

- Declare an empty object
- Add properties with *dot notation*

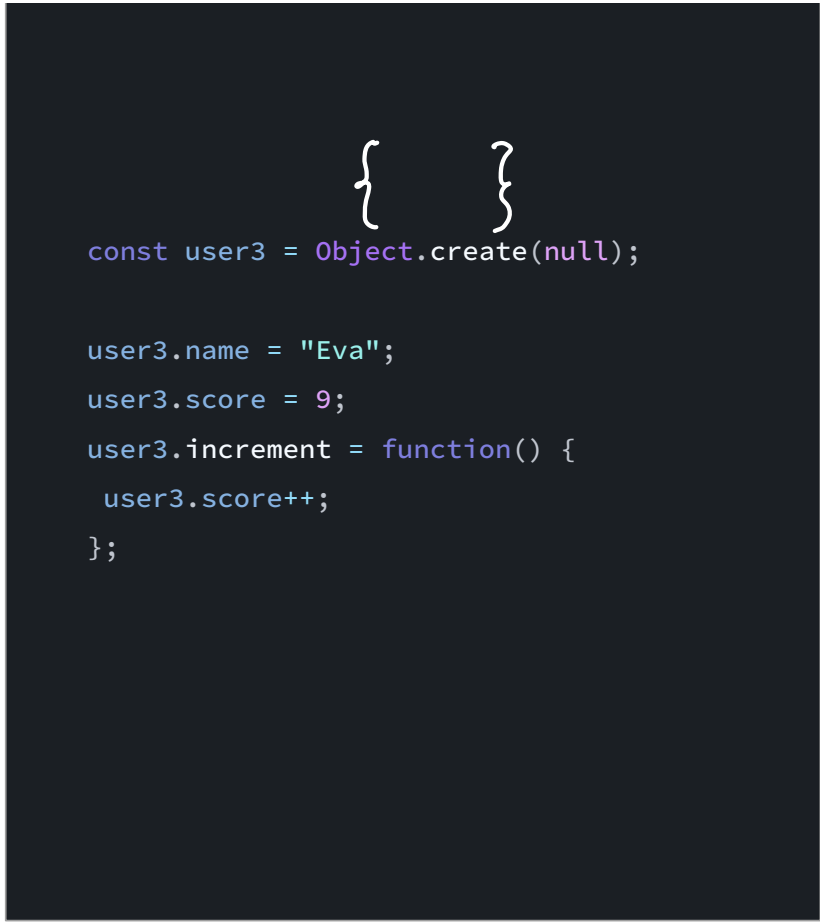
```
const user2 = {}; //create an empty
object

//assign properties to that object
user2.name = "Jason";
user2.score = 6; 7
user2.increment = function() {
 user2.score++;
};

user2.increment(); //user2.score => 7
```

Creating user3 using *Object.create*

- *Object.create* is going to give us fine-grained control over our object later on

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

And suppose we have millions of users! What could we do?



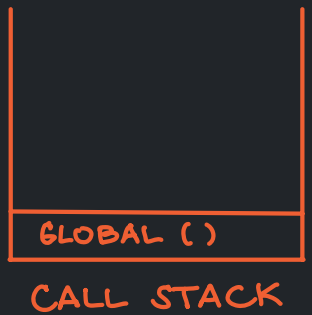
Solution 1. Generate objects using a function

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


const user1 = userCreator("Alex", 3);
const user2 = userCreator("Jason", 6);
user1.increment()
```

Global Execution Context

```
1  function userCreator(name, score) {
2    const newUser = {};
3    newUser.name = name;
4    newUser.score = score;
5    newUser.increment = function() {
6    newUser.score++;
7    };
8    return newUser;
9  };
10
11 const user1 = userCreator("Alex", 3);
12 const user2 = userCreator("Jason", 6);
13 user1.increment()
```

TOE



GLOBAL MEMORY

userCreator = f

user1 = { name: "Alex", score: 3 4, inc: f }

user2 = { name: "Jason", score: 6, inc: f }

Solution 1. Generate objects using a function

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!

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


const user1 = userCreator("Alex", 3);
const user2 = userCreator("Jason", 6);
user1.increment()
```

Solution 2: Using the prototype chain

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

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

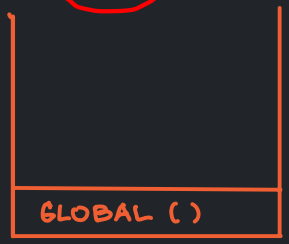
Make the link with `Object.create()` technique

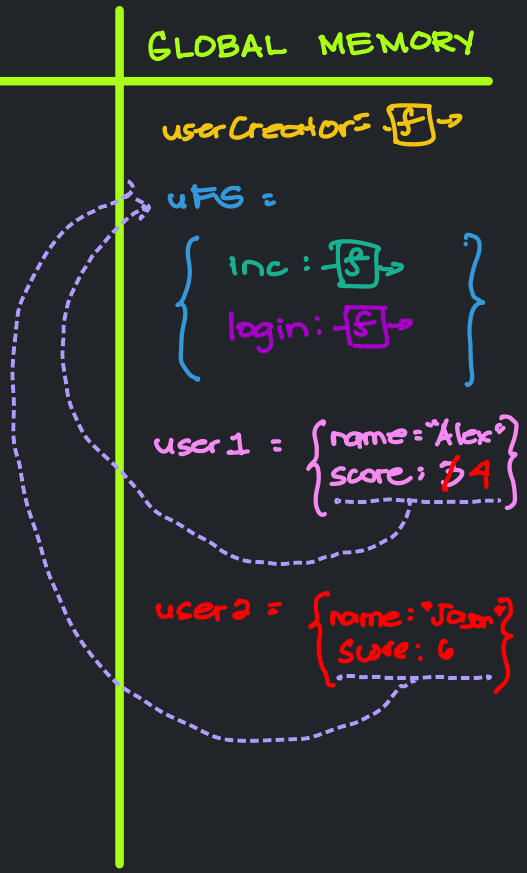
```
function userCreator (name, score) {
  const newUser = Object.create(userFunctionStore);
  newUser.name = name;
  newUser.score = score;
  return newUser;
};

const userFunctionStore = {
  increment: function(){this.score++;},
  login: function(){console.log("Logged in");}
};

const user1 = userCreator("Alex", 3);
const user2 = userCreator("Jason", 6);
user1.increment();
```

Global Execution Context

```
1  function userCreator (name, score) {
2    const newUser = Object.create(userFunctionStore);
3    newUser.name = name;
4    newUser.score = score;
5    return newUser;
6  };
7
8  const userFunctionStore = {
9    increment: function(){this.score++;},
10   login: function(){console.log("Logged in");}
11 };
12
13 const user1 = userCreator("Alex", 3);
14 const user2 = userCreator("Jason", 6);
15 user1.increment();
```

TOE





Solution 2: Using the prototype chain

Problem

No problems! It's beautiful. Maybe a little long-winded

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

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

Super sophisticated but not standard

```
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("Logged in");}
};

const user1 = userCreator("Alex", 3);
const user2 = userCreator("Jason", 6);
user1.increment();
```

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

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?

```
const user1 = new UserCreator("Alex", 3)
const user2 = new UserCreator("Jason", 6)
```

The ***new*** keyword automates a lot of our manual work

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?

```
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("Alex", 3);
```

Automates the hard work

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

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

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

Interlude - functions are both objects and functions 🤨

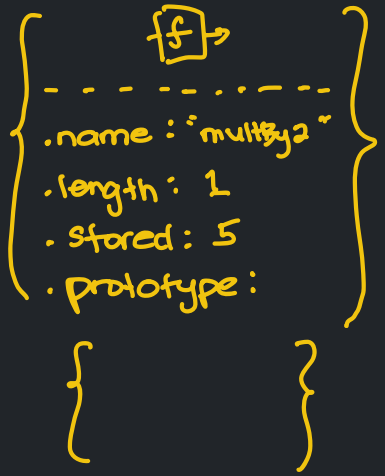
We could use the fact that all functions have a default property on their object version, ***prototype***, which is itself an object - to replace our ***functionStore*** object

Global Execution Context

```
1  function multiplyBy2(num){
2    return num*2;
3  };
4
5  multiplyBy2.stored = 5;
6  multiplyBy2(3); // 6
7
8  multiplyBy2.stored; // 5
9  multiplyBy2.prototype; // {}
```

GLOBAL MEMORY

multBy2 =



Complete Solution 3: Using the ***new*** keyword

```
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()
```

Global Execution Context

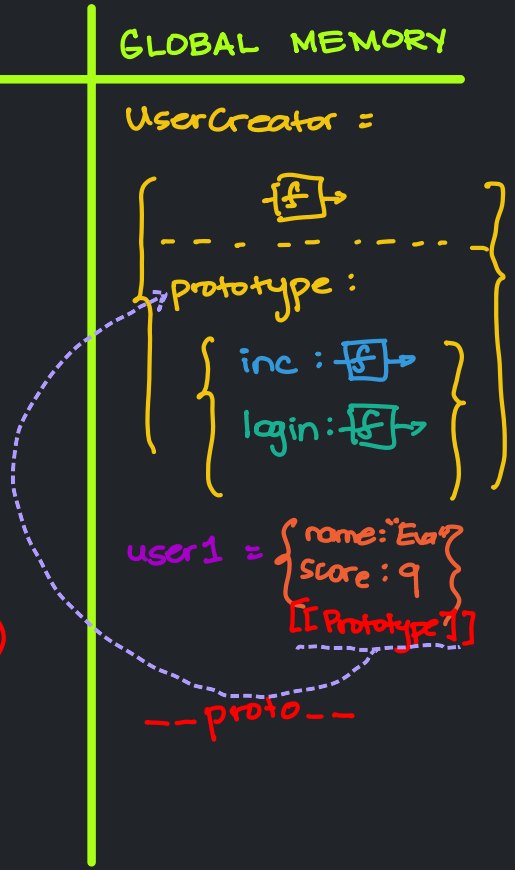
```
1   function UserCreator(name, score){
2     this.name = name;
3     this.score = score;
4   };
5
6   UserCreator.prototype.increment = function(){
7     this.score++;
8   };
9   UserCreator.prototype.login = function(){
10  console.log("login");
11  };
12
13  const user1 = new UserCreator("Eva", 9)
14
15  user1.increment()
```

TOE

console.dir (obj/func)

console.log (user1.__proto__)

Object.getPrototypeOf



Complete Solution 3: Using the ***new*** keyword

Benefits:

- Faster to write
- Still typical practice in professional code

Problems:

- 95% 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!

```
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: The ***class*** 'syntactic sugar'

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

```
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();
```

Solution 4: The ***class*** 'syntactic sugar'

```
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()
```

```
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();
```

Solution 4: The ***class*** 'syntactic sugar'

Benefits:

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

Problems:

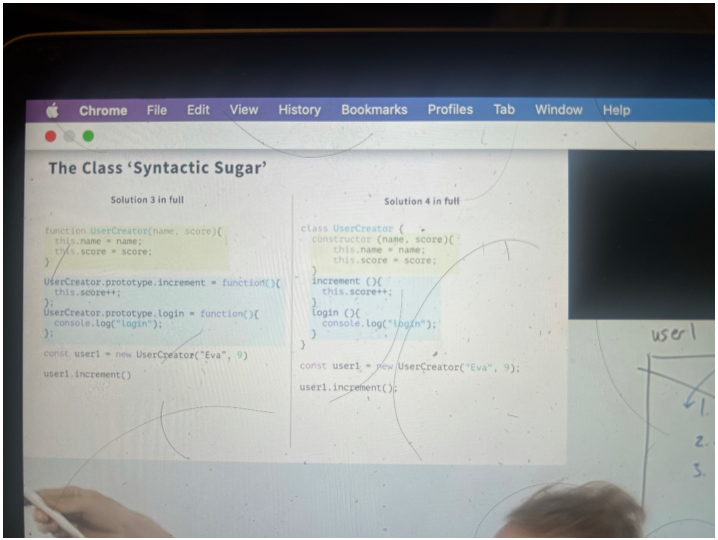
- 99% of developers have no idea how it works and therefore fail interviews
- But you will not be one of them!

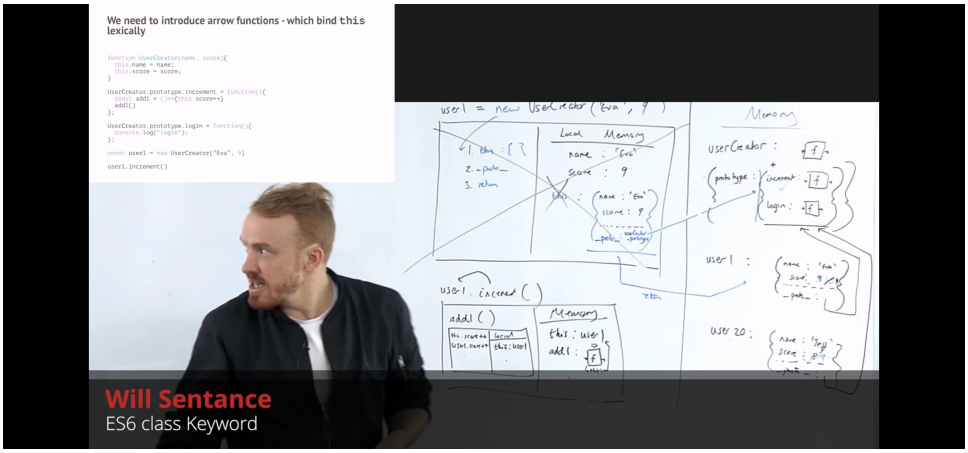
```
class UserCreator {
 constructor (name, score){
    this.name = name;
    this.score = score;
 }
 increment (){
   this.score++;
 }
 login (){
   console.log("login");
 }
}

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

user1.increment();
```

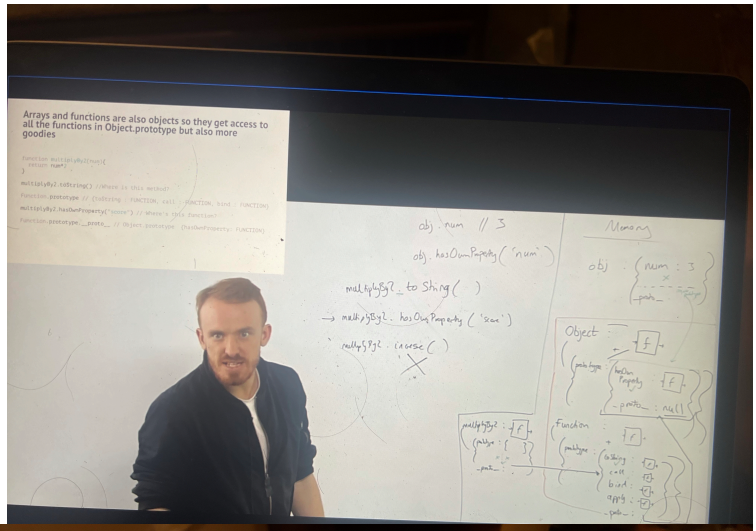
Further Reading

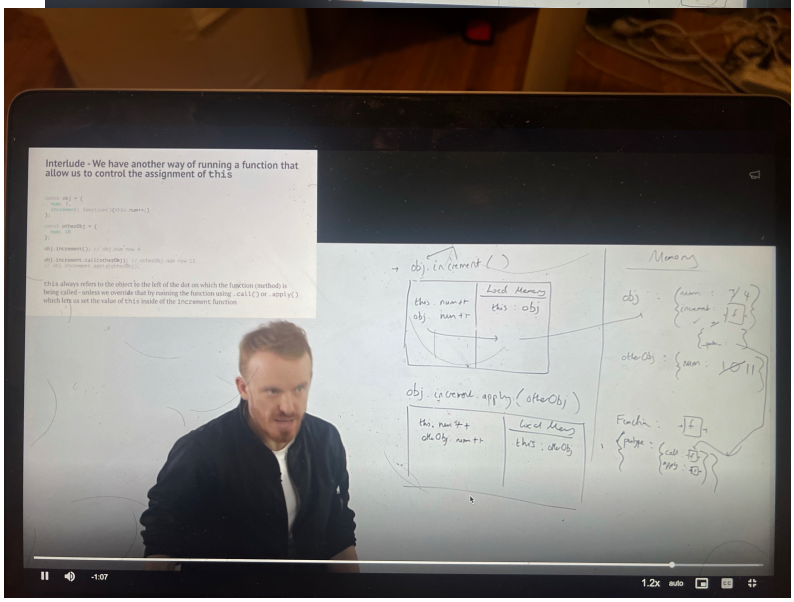
[Understanding subclassing in javascript: 'extends' and 'super']

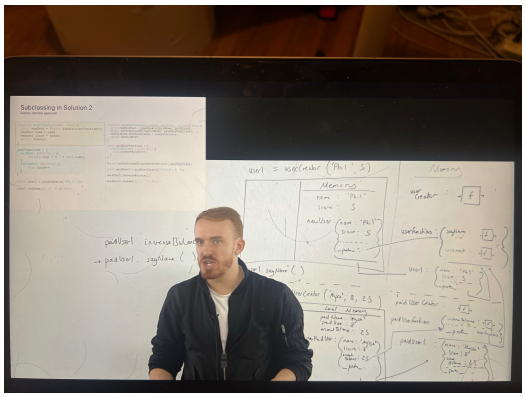
[Prototypes in Javascript]

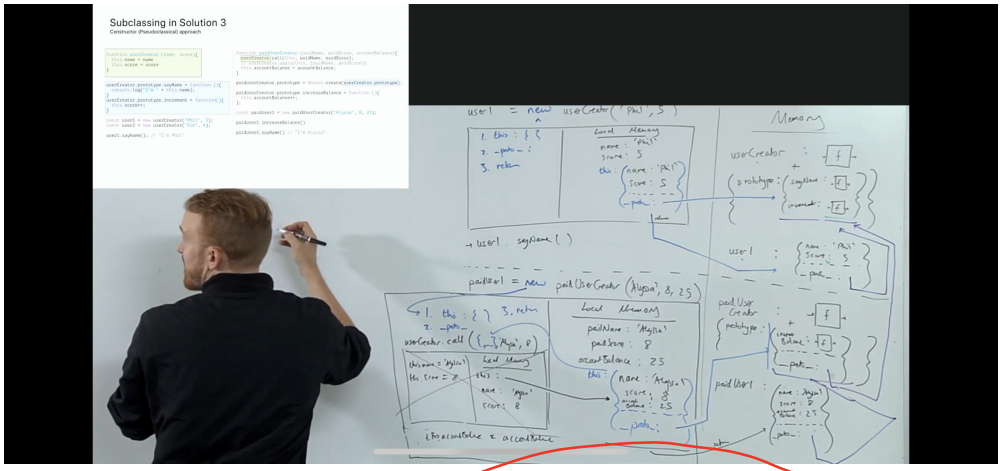


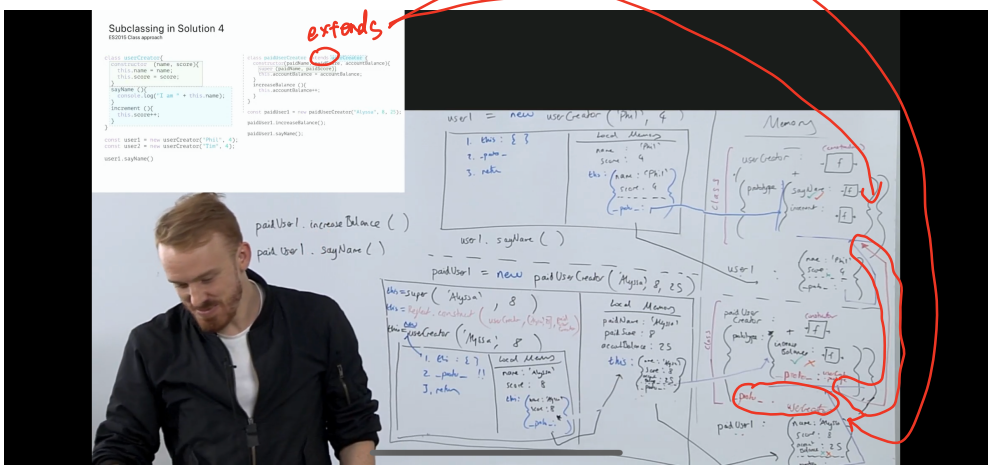

We need to introduce arrow functions - which bind this lexically
Will Sentance
ES6 class Keyword


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


Interlude - We have another way of running a function that allow us to control the assignment of this
-1:07
1.2x
auto


Subclassing in Solution 2


Subclassing in Solution 3
Constructor (Pseudoclassical) approach


Subclassing in Solution 4
ES2015 Class approach
extends