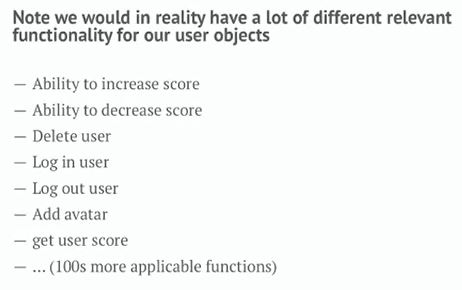


The best way to store the above data would be to store it in an object, because an object allows binding the data with the associated functionality.



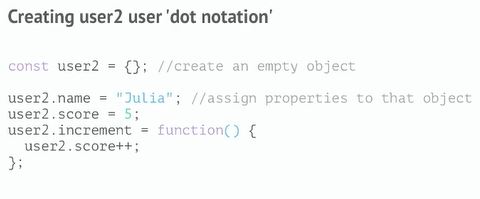


## What alternative techniques do we have for creating objects?

### Solution 1:



### Solution 2:

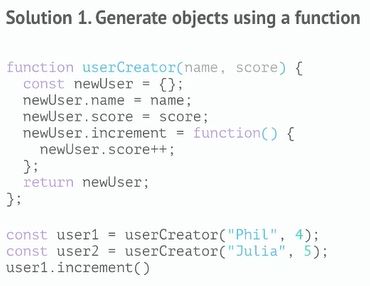


### Solution 3:

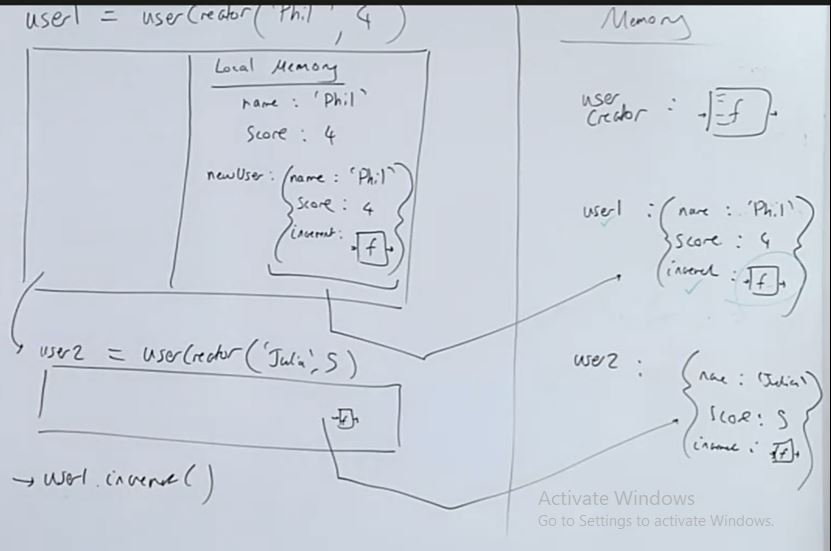


As our code is getting repetitive it is good to have a function which we can call to generate the object

### Solution 1 (Functions):



Now let us visualize what is happening behind the scenes:

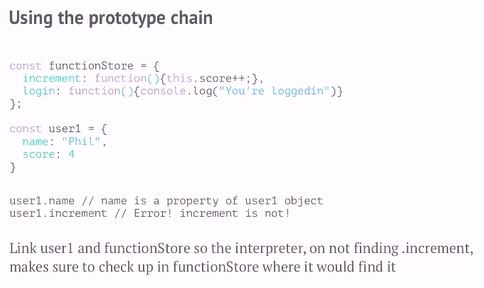


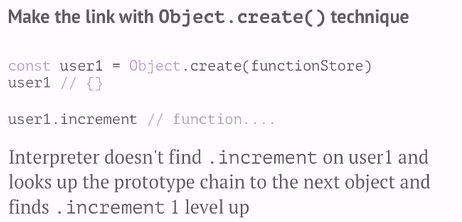
The problem with this approach is that we are creating a new function every i.e. the increment function every time we create a new object.

### Solution 2 (Object.create()):

In this solution, we 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

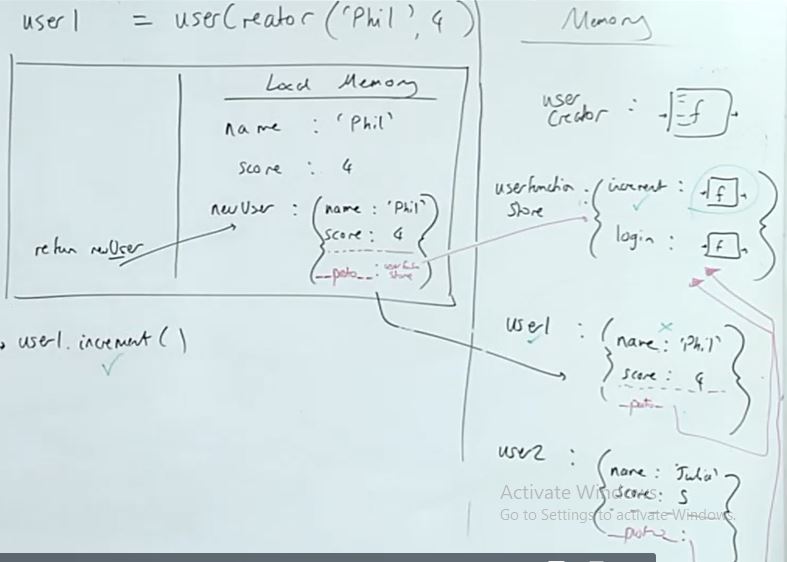
How to make this link?





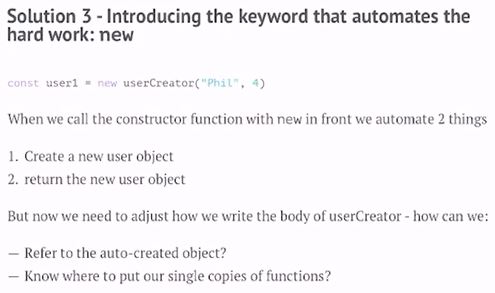


Let’s look at how the solution 2 works under the hood:



Here the user object somehow has access to the increment and the login method, and we can see that it is because of the hidden property associated with the object that allows us to associate the functions increment and login with the user through the Object.create() method. One of the hidden properties associated with any object in JavaScript is the \_\_proto\_\_ property. Here the Object.create() will associate the \_\_proto\_\_ property of the current user object with the userFunctionStore. Here if we call the increment() method on the user1, initially it will look inside the user1 object but it won’t be able to find it, subsequently it will look for the increment() in the \_\_proto\_\_ property which points to the userFunctionStore which contain the increment() property.

### Solution 3 (new):



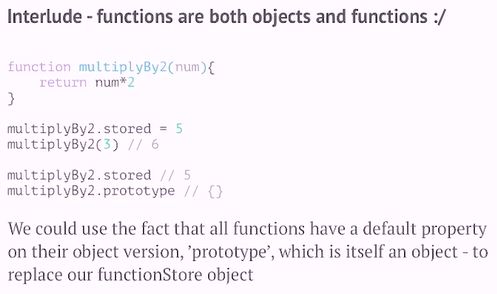
So the object creation and returning an object is automatically handled by **this** keyword.

Q) But how we refer the auto-created object?

A) In the solution 2 inside the function we referred to the current object with the userCreator. But in the **new** keyword we will refer to the current object using **this** keyword.

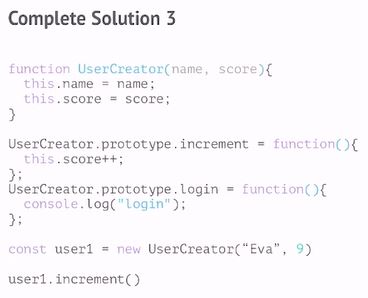
Q) How do we link functions that in solution 2 we used Object.create() which linked the \_\_proto\_\_ property to the function given as input?

A) Javascript gives us a place where we can put this function which will be automatically linked to the \_\_proto\_\_ property of objects created using the new keyword, and that place can be understood by understanding the concept given below:

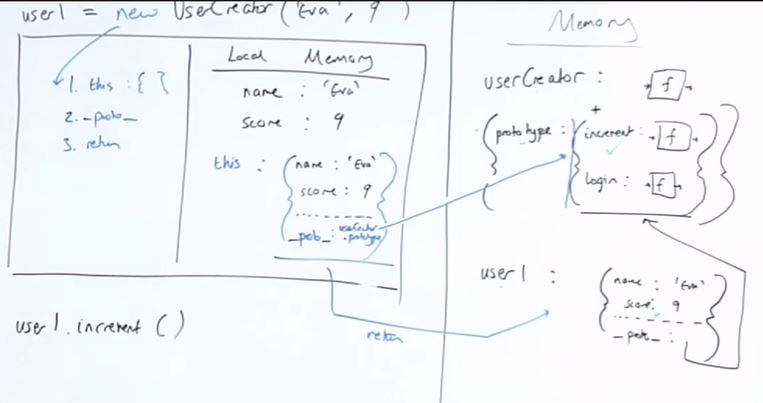


So here as we can see that all the function have an object version of themselves which by default contain a **prototype** property where we can store all of the common function that are shared among all the objects.

In a nutshell, we can access the function part using the parenthesis and the object part using the dot notation of a function.



Just remember, we will store the single copies of our function to the prototype property of the object part of our UserCreator function in the above example. Let us look at what is happening behind the scenes:

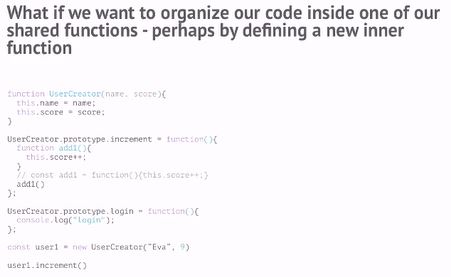


As we can see the **new** keyword initially creates an empty object and assigns **this** keyword to it. Thereafter it points the \_\_proto\_\_ property to the prototype property of the userCreator function’s object part. Finally the new keyword will return the object to the user1

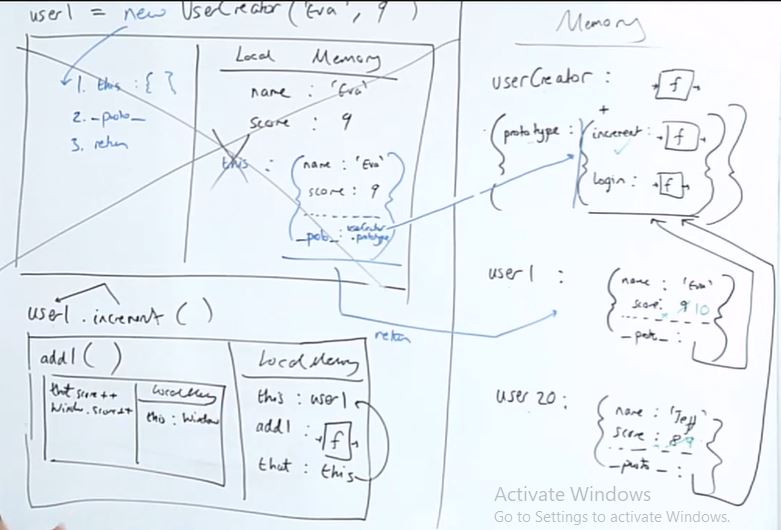
**This** keyword:

The **this** keyword inside a regular function() will refer to the global **this** object which points to the window function. The **this** keyword inside a function which is called after the **new** keyword points to the empty object which is used by the new keyword and attaches some values and returns it which is all done by the **new** keyword. The **this** keyword inside a function which is called with the dot operator ex: user1.increment() will always refer to the object on the left i.e. user1

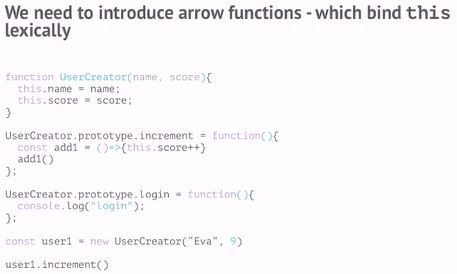
Let us look at the code below with a slight variation to understand a concept:



Let us look at what is happening behind the scenes:

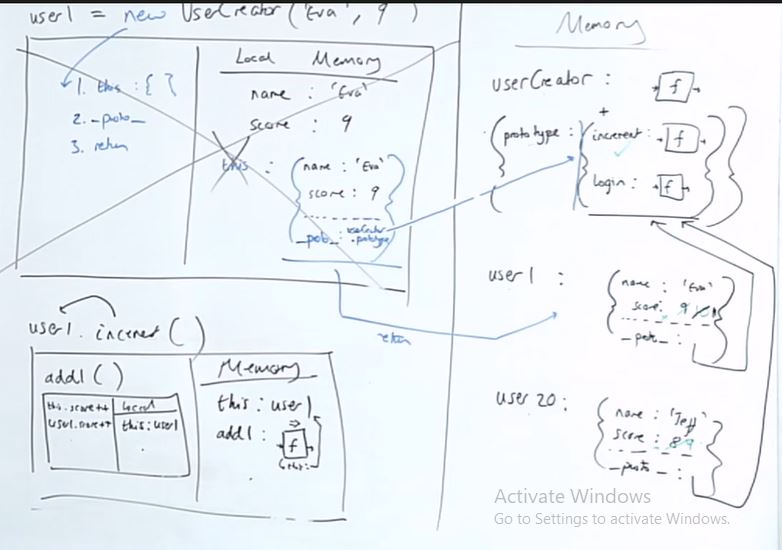


Here as we can see from the image above that the **this** assignment inside the add() function refers to the window object but we do not want that rather we want this to refer to the user1, so we will use the arrow function which will preserve the this reference inside the add function as given below:

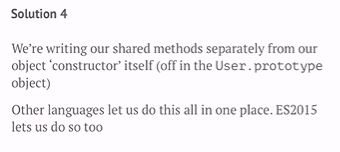


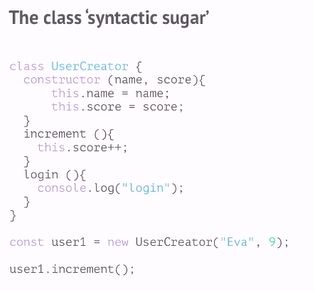
So by using the arrow function **this** will be lexically scoped. So **this** inside the arrow function will refer to what **this** was outside the arrow function.

Let us visualize this:

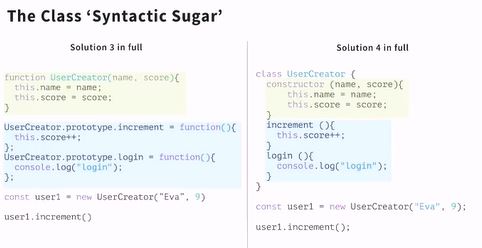


### Solution 4 (class):



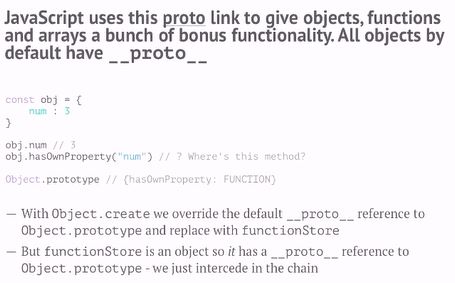


Let us compare both solutions:

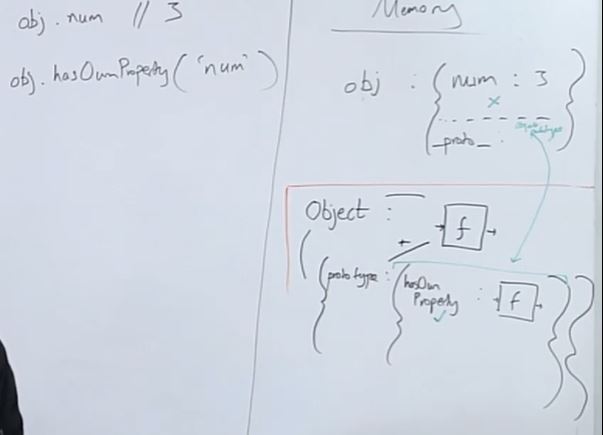


Here any function declared inside the class will be automatically defined in the prototype property of the object part of UserCreator.

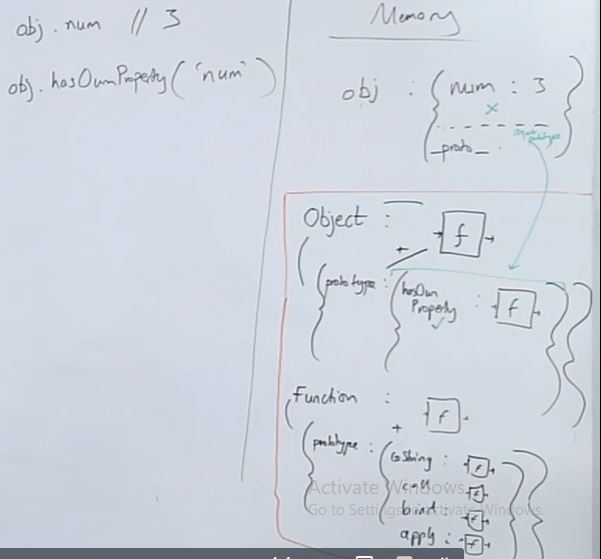
Now we will see that JavaScript has number of functions available to it via the \_\_proto\_\_ property which refers to Object.prototype

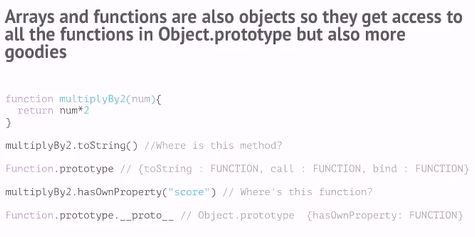


Let us visualize how this works under the hood:

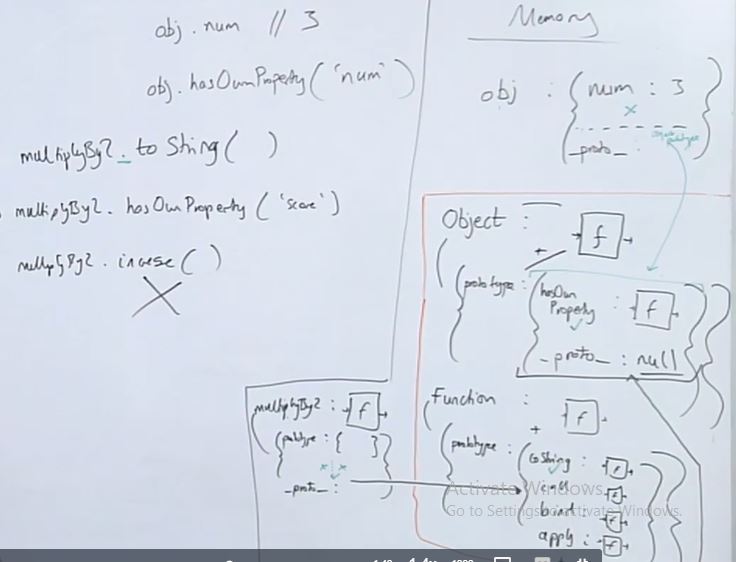


When the JavaScript runtime loads it gives us with some default methods, which we can utilize to make our code more efficient and adds more functionality to our code. JavaScript gives us Object and Function. As all the Object part in the Function/Object combo has the prototype property, the top of the chain parent Object’s prototype property includes some default methods like hasOwnProperty() and so on. While the top of the chain Function’s prototype property includes some default methods like toString(), call(), bind(), apply() and so on as given below:





Let us visualize this below in order to see what is happening:



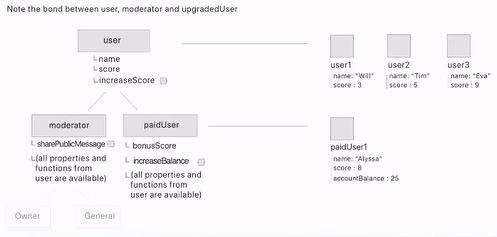
Here in the code above the line multiplyBy2.toString() will access the **object** part as it uses the **dot** notation. As it will not find the **toString** function it will look up in its **\_\_proto\_\_** property which first look into the JS provided **Function** base object’s prototype property which infact contains the **toString()** function.

Subsequently, now if we try to access the **hasOwnProperty()** the JS will first look in the object’s memory it will not find it there so it will look it in the **\_\_proto\_\_** property which points to the **Function** but it won’t find it in the **Function’s** prototype as well so it will look into the **Function \_\_proto\_\_** property which will point to the base **Object’s** prototype property where it will be able to find **hasOwnProperty()** and execute it

**Note**: the **\_\_proto\_\_** property of the base **Object** will be **null** as it the parent and the prototype chain ends there.

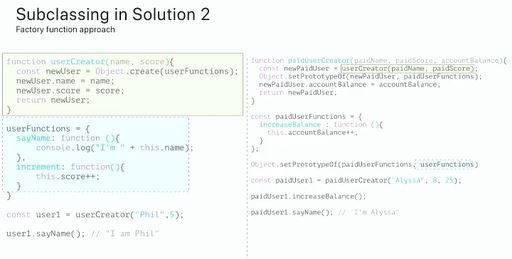
## Subclassing:

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

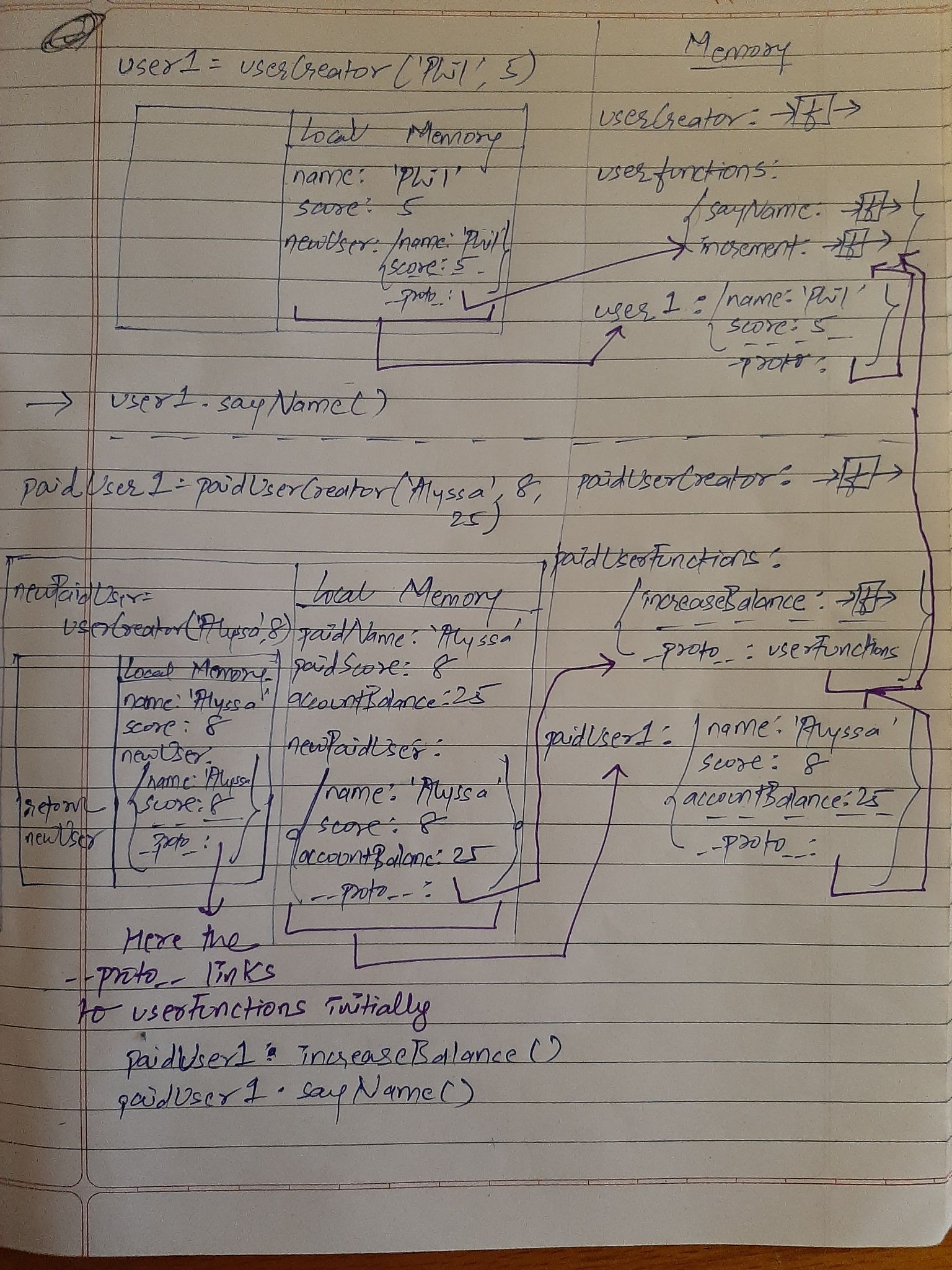


We can achieve the classes inheritance functionality with javascript using all of the three solutions 2, 3, and 4 as discussed above

### Solution 2:

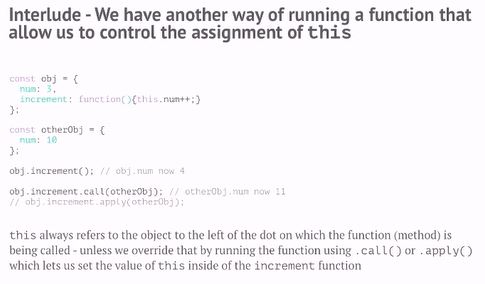


So here we are using the factory function approach to achieve the inheritance. Let us look at what is happening behind the scenes and discuss it in order to gain understanding:

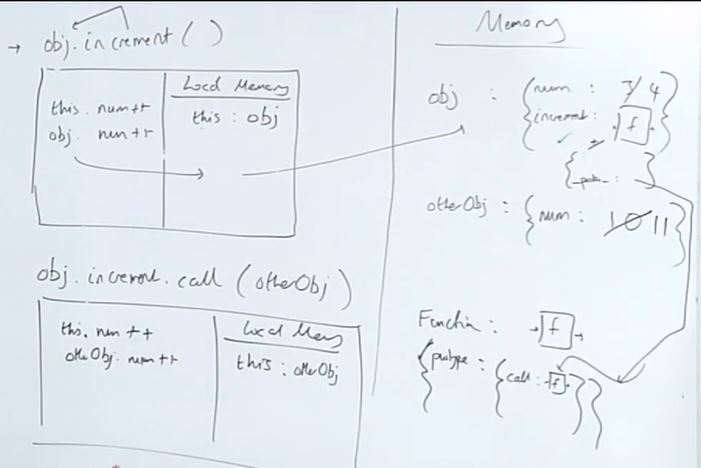


Object.setPrototypeOf(user1, user2) - The setPrototypeOf will control the \_\_proto\_\_ of user1 and set it to point to the user2. Initially the newPaidUser function’s \_\_proto\_\_ property in the paidUserCreator() function in the code above will point to the userFunctions, but after executing the line Object.setPrototypeOf(newPaidUser, paidUserFunctions) the \_\_proto\_\_ will point to the paidUserFunctions.

### Controlling **this** reference (Interlude):



The **call()** and **apply()** function will allow us to manually control what **this** inside the function refers to. In order to understand Solution 3 we need to understand how the **call()** and **apply()** method works under the hood.



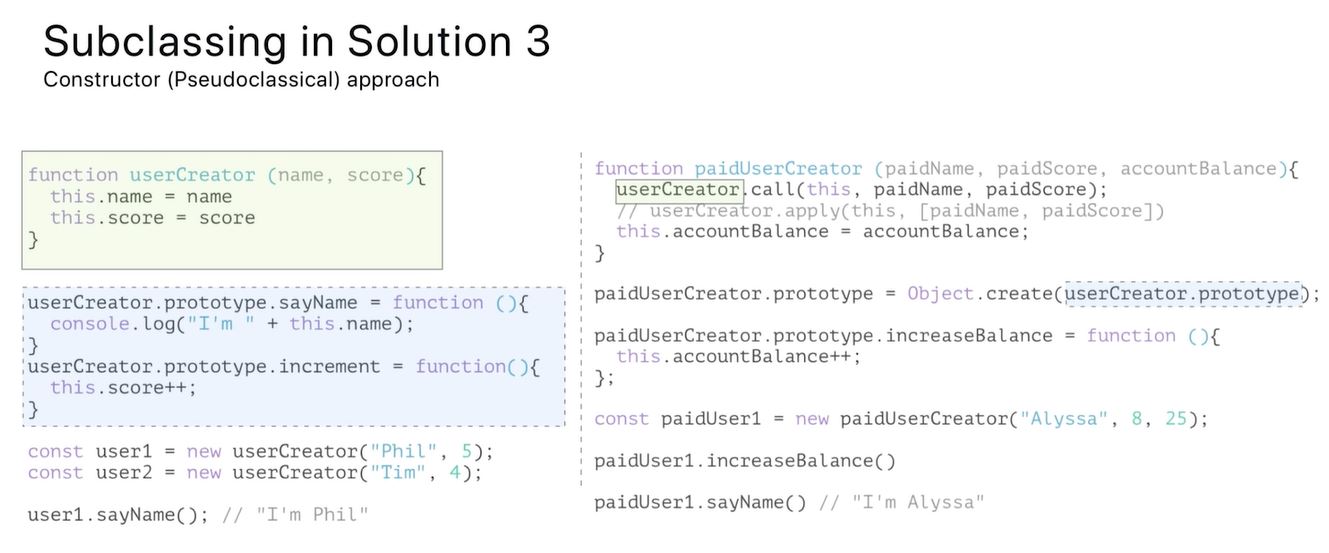
Here the **call** method is available to the increment method via the **object** part of the **function + object** combo of the **increment** function whose **object** part contains the **hidden \_\_proto\_\_ property** which points to the **Function** which in fact contains the **prototype** property where the JS defines the **call** method.

So **this** inside the increment method will now refer to the **otherObj** as we have used the **call** method

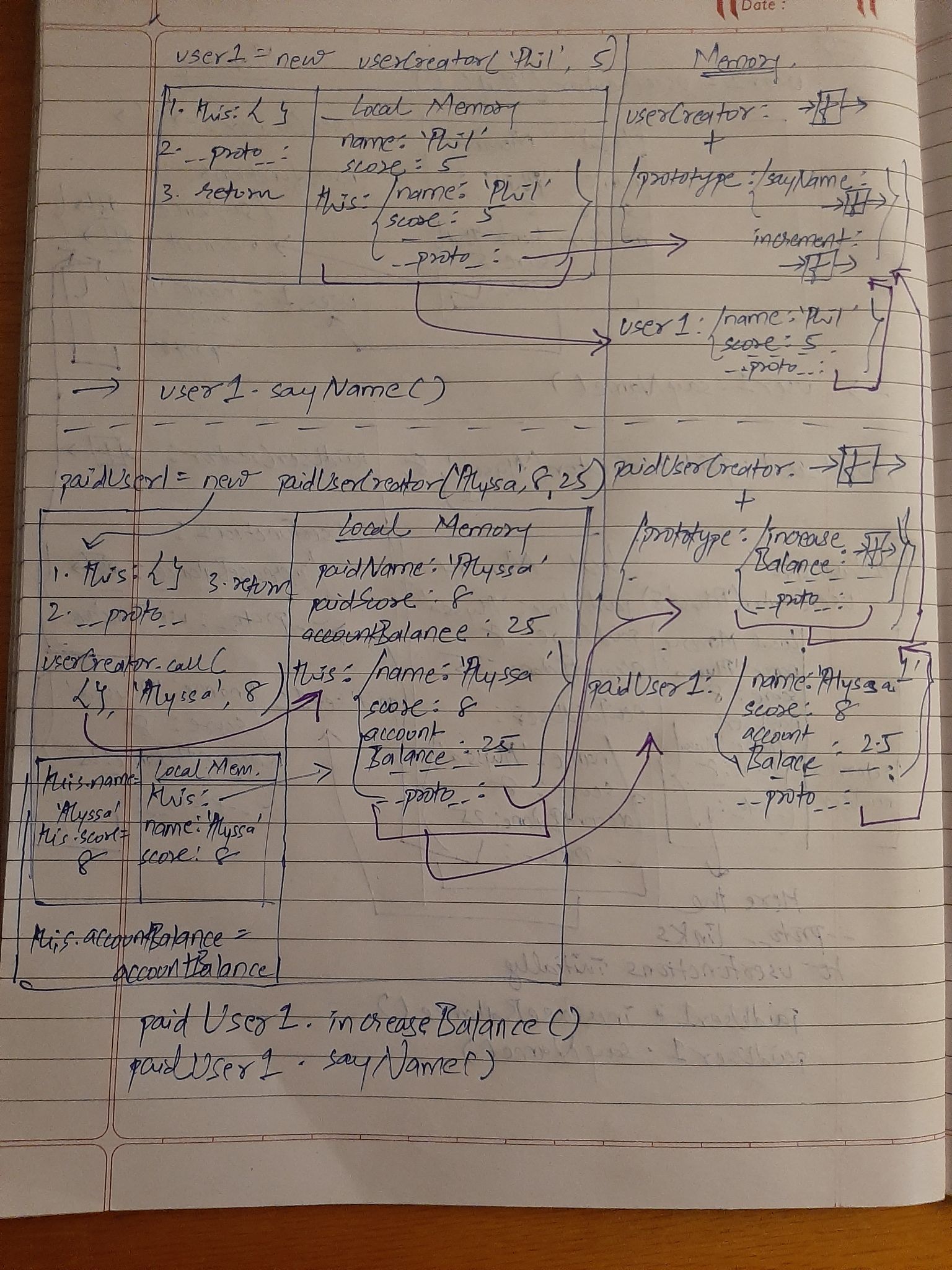
### Difference between call() and apply():

The difference between **call()** and **apply()** is that if our increment function were to take any arguments and we intend to use the **call()** method then the argument in the increment function will be passed as comma separated value in the **call()** function, with the first argument of the **call()** function obviously being the reference of **this** followed by those comma separated values. On the other hand, in the **apply()** the first argument is obviously the **this** reference value while the second argument is the array which contains the comma separated values/parameters to be used by the **increment** function.

### Solution 3 (using the call() method to achieve inheritance):



Let us look at how the code above works under the hood:

****

For line: **paidUserCreator.prototype = Object.create(userCreator.prototype);**

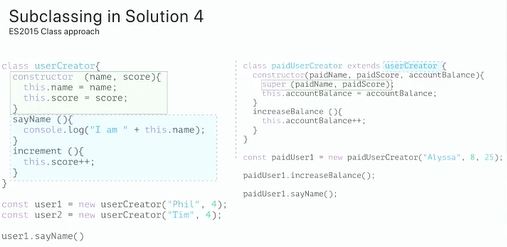
Here, we are overwriting the default **prototype** property of the **object** part of the **paidUserCreator** function and setting it to an **empty** object whose **\_\_proto\_\_** property points to the **userCreator’s** prototype using the **Object.create()**

For line: **userCreator.call(this, paidName, paidScore);**

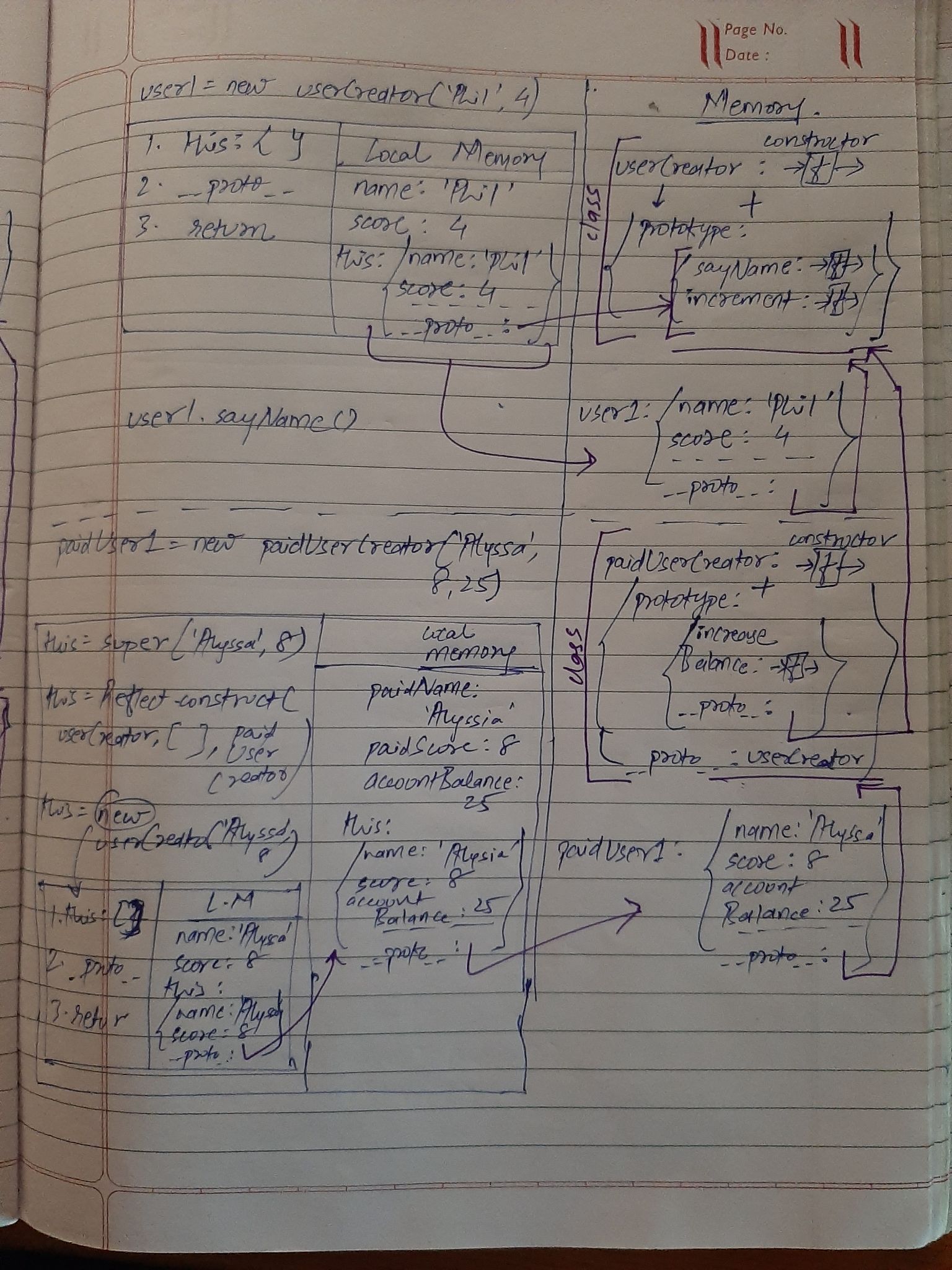
Here, we are making sure that **this** inside the call refers to **this** i.e. first parameter which in fact refers to the outer this which is automatically created by the **new** keyword. So basically we are not associating **this** to the **userCreator** instead we are referring it to **this** from one level above.

Now let us look at our final approach:

### Solution 4 (using class and extends):



Let us look at how it works behind the scenes:



Here what **extends** keyword does is essentially linking the **\_\_proto\_\_** property of the **paidUserCreator** function’s **object’s prototype** part of the **function + object** combo to the **prototype** property of the **userCreator** function’s **object** part to get access to all its properties

Another thing that **extends** does is set the **\_\_proto\_\_** property of the **paidUserCreator** function’s **object** part of the **function + object** combo to the **userCreator**

So here the fundamental thing that changes from the Solution 4 is that in Solution 3 we used to pass **this** of the current context inside the **call** function in order to ensure that **this** in the **userCreator** refers to **this** in the **paidUserCreator** and it changes that outer **this**. But, here we are not instantiating **this** to anything and directly calling the **super()** method. So to think it logically we can imagine **this** being assigned to the return value of calling the **super()** method. Furthermore, the **super()** method will run the constructor function of the **userCreator** by looking into the **\_\_proto\_\_** value of the **paidUserCreator** function’s **object** part of the **function + object** combo.

Now while running the constructor function of the **userCreator**, **super()** will set the **\_\_proto\_\_** property of the **object** inside that constructor function to the **paidUserCreator’s** prototype property.