Module3

Exercise4(ES6 Part-1)

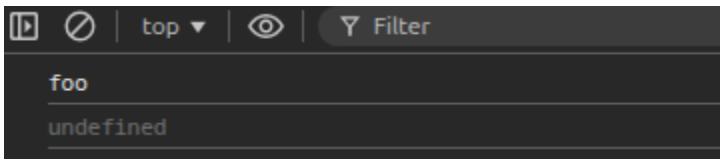
Q 1) Predict the output and Explain

```
 const obj = {
        a: "foo",
        b: function () {
          console.log(this.a);
        },
};

const c = obj.b;

obj.b();
c();
```

A1.) The output of the first function call when we are calling the function with the object
obj.b() gives the output as “foo”
And the second function call with the reference variable const c=obj.b
c(); gives us undefined

The different output even after calling the same function is because that the C only has the reference function key not the this itself when it is called it basically looking for name within his context using this.

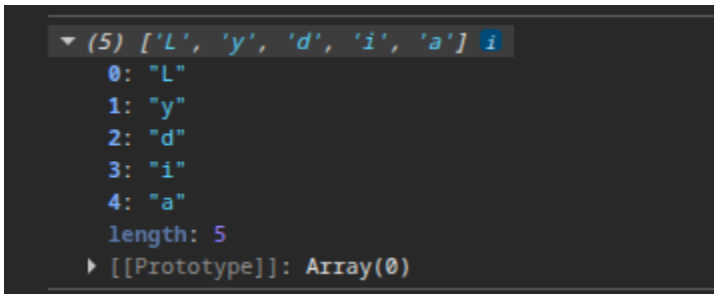
For resolving this issue we will use .bind() to permanently bind the c with the object context.



2) output of below and Explain

[...'Lydia'];

A2.) In the above question the **"..."** spread operator is used for expanding any iterable in individual element. In the above question the string will get expanded into character forming a array of character containing 5 characters array will form due to the square bracket. If we try to console it we will get ["L","y","d","i","a"]



3)Predict the output and Explain

let user={ name:"Piyush", age:24 }

const {name}= user;

console.log(name);

const { name: myName } = { name: 'Lydia' };

console.log(name);

A3.) In the above question we at the first console.log(name) we will get the value of name key from user inside name variable giving us Piyush .

It is due to the destructuring introduced in ES6 where we can assign the values from key of an Object to a variable using simple code
let {key}=object;

Now for the second console.log(name) after the destructuring like this

const {name:myName} ={name:'Lydia'}
We will still get Piyush It is due to that we have are trying to assign Lydia to name but we are assigning it to myName because in destructuring we can destructure a object in two way By using there as a identifier or if that identifier is already in use we can use this format {key:newVar}=user;

So if we try to access the name we will get Piyush and myName we will get Lydia.

4) Predict the Output and Explain
console.log({a:1} == {a:1});
console.log({a:1} === {a:1});

A4.) In JavaScript, objects are compared by reference, not by value.

{ a: 1 } creates a new object in memory each time it is written.

Even though both objects have the same structure and values, they are stored at different memory locations.

Therefore, both objects have different references.

== (loose equality)

For objects, == does not compare values.

It checks whether both variables refer to the same object in memory.

Since the references are different, the result is false.

=== (strict equality)

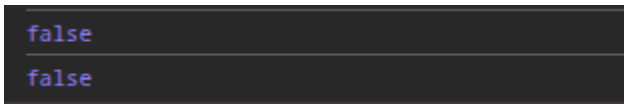
=== also compares objects by reference, not by content.

It additionally checks type, but in this case both operands are objects.

Because the memory references are still different, the result is false.

Two objects are equal only if they reference the same memory location, not if they merely look the same.

So both the console.log will ge false as an output



5) Predict the output and Explain
let person = { name: 'Lydia' };
const members = [person];
person = null;

console.log(members);

A5.) In this question when we are trying to console.log(members) we will get an array of object containing a single key value pair person

This is due to the referencing of the object person to the members[0].

Here what we are basically doing that we are creating an object person and assigning it to the members array but as we know the object are pointed as memory reference as here the person is point to the object.

Then when we are assigning it to members we are basically members[0] to object reference at memory and when we are changing the person value as null we are only changing its reference to null not the value.

Therefore when we try to access the members we gets the object instead of null because person points to null not the members.


[{…}]
0: {name: 'Lydia'}
length: 1
[[Prototype]]: Array(0)
Array(0)