Exercise on Proxies

Using your knowledge of proxies object, you must use them to:
i) log the number of times a function is accessed
ii) create a revocable discount object

Exercise 1:

Suppose the following `fibonacci` implementation is given:

```
fibonacci = n =>
    n <= 1 ? n :
    fibonacci( n - 1 ) + fibonacci( n - 2 );
```

Determine how many times the `fibonacci` function is called when evaluating `fibonacci( 12 )`.

Determine how many times `fibonacci` is called with the argument `2` when evaluating `fibonacci( 12 )`.

```
let fibonacci = n =>
   n <= 1 ? n : fibonacci( n - 1 ) + fibonacci( n - 2 ); //the fibonacci function

let fibCalls = 0; // total calls to the fibonacci function
let fibCallsWith2 = 0; // calls to the fibonacci function with 2 as an argument

//Write your code here



// Do not edit code below this line
console.log( 'fibCalls:', fibCalls );
console.log( 'fibCallsWith2:', fibCallsWith2 );
```

Explanation:

The solution is not that hard. As we learned earlier, `fibonacci = new Proxy( fibonacci, {` uses the same reference name as our fibonacci function so that

the proxy can be applied on all recursive calls as well.

Within the proxy, we increment `fibCalls` and `fibCallsWith2` whenever needed.

`return Reflect.apply( target,thisValue,args);` could also be written as `return target(...args);`

Exercise 2:

Create a proxy that builds a lookup table of `fibonacci` calls, memorizing the previously computed values. For any `n`, if the value of `fibonacci( n )` had been computed before, use the lookup table, and return the corresponding value instead of performing recursive calls.

Use this proxy, and determine how many times the `fibonacci` function was called

- altogether
- with the argument `2`

while evaluating `fibonacci( 12 )`.

```
let fibonacci = n =>
   n <= 1 ? n : fibonacci( n - 1 ) + fibonacci( n - 2 );// the fibonacci function

let fibCalls = 0; // total calls to the fibonacci function
let fibCallsWith2 = 0; // total calls to the fibonacci function with 2 as an argument

//Write your code here


//Do not edit the code below this line

console.log( 'Result:', fibonacci( 12 ) );
console.log( 'fibCalls', fibCalls );
console.log( 'fibCallsWith2', fibCallsWith2 );
```

Explanation:

This is very similar to what we did in exercise 1. The only difference is that we

now use a lookup table to see if we already have the Fibonacci value for a certain number. If it doesn't exist, we add it to the table.

There are several ways to implement your table I have used `Map()` because of it's simplicity. The rest is simple code where we increment our `fibCalls` and `fibCallsWith2` variables.

Then comes the lookup functionality, where we check if the key from `args[0]` already exists in the table.

Exercise 3:

Given the object

```
let course = {
    name: 'ES6 in Practice',
    _price: 99,
    currency: '€',
    get price() {
        return `${this._price}${this.currency}`;
    }
};
```

Define a revocable proxy that gives you a 90% discount on the price for the duration of 5 minutes (or 300,000 milliseconds). Revoke the discount after 5 minutes.

The original `course` object should always provide access to the original price.

Exercise 3 | Solution

```
let course = {
    name: 'ES6 in Practice',
    _price: 99,
    currency: '€',
    get price() {
        return `${this._price}${this.currency}`;
    }
};

//Write your Code here
```

Explanation:

We create a revocable proxy `revocableDiscount`. In the `get` function, we check if our key to the target is 'price':

```
if ( key == 'price' )
```

We then set the new discounted price and return it.

`return target[ key ]` makes sure that other properties of the `course` object are returned as they are.

Now all that's left is to set the delay after which the proxy is to be revoked.

This is done using `setTimeout`.