Defining Proxies

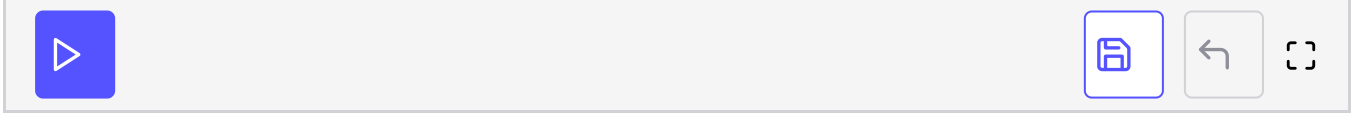
solve fundamental operations (like property lookup, assignment, enumeration, function invocation, etc.) using proxies

We can create a proxy in the following way:

```
let proxy = new Proxy( target, trapObject );
```

The first argument is `target` representing the proxied constructor function, class, or object.
The second argument is an object containing traps that are executed once the proxy is used in a specific way.
Let's put our knowledge into practice by proxying the following target:

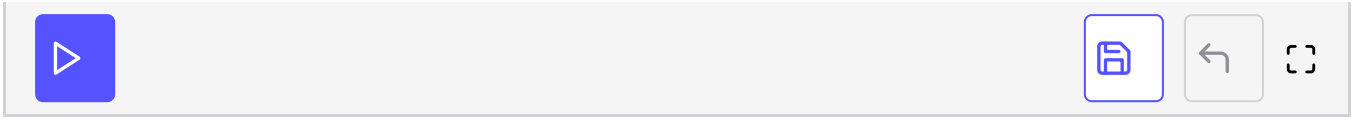
```
class Student {
    constructor(first, last, scores) {
        this.firstName = first;
        this.lastName = last;
        this.testScores = scores;
    }
    get average() {
        let average = this.testScores.reduce(
            (a,b) => a + b,
            0
        ) / this.testScores.length;
        return average;
    }
}

let john = new Student( 'John', 'Dwan', [60, 80, 80] );
console.log(john);
```



We will now define a proxy on the target object `john`:

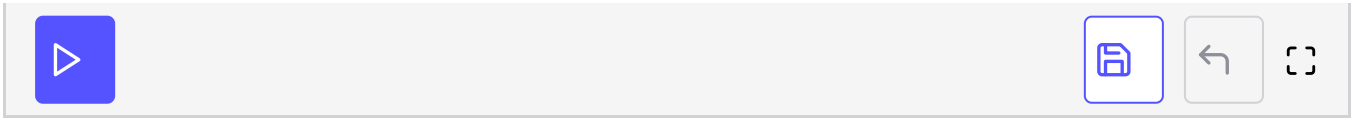
```
let johnProxy = new Proxy( john, {
    get: function( target, key, context ) {
        console.log( `john[${key}] was accessed.` );
```

```
        // return undefined;
    }
});
```



The `get` method of the proxy is executed whenever we try to access a property.

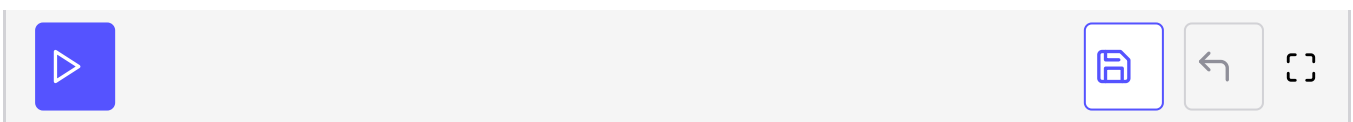
```
console.log(johnProxy.getGrade);
//> john[getGrade] was accessed.
//> undefined

console.log(johnProxy.testScores);
//> john[testScores] was accessed.
//> undefined
```



The above defined `get` trap chose to ignore all the field values of the target object to return `undefined` instead.

Let's define a slightly more useful proxy that allows access to the `average` getter function, but returns `undefined` for anything else:

```
let johnMethodProxy = new Proxy( john, {
    get: function( target, key, context ) {
        if ( key === 'average' ) {
            return target.average;
        }
    }
});

console.log(johnMethodProxy.firstName);
//undefined
console.log(johnMethodProxy.average);
//73.33333333333333
```

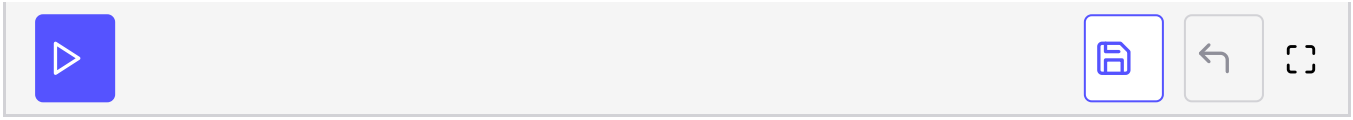


We can conclude that proxies can be used to

- define access modifiers,
- provide validation through the public interface of an object.

The target of proxies can also be functions.

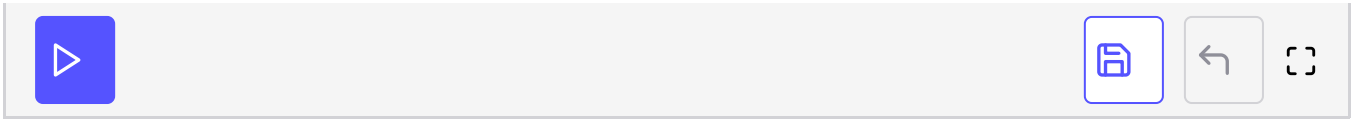
```
let factorial = n =>
    n <= 1 ? n : n * factorial( n - 1 );
console.log(factorial(6));
```



For instance, you might wonder how many times the `factorial` function was called in the expression `factorial(5)`. This is a natural question to ask to formulate an automated test.

Let's define a proxy to find the answer out.

```
factorial = new Proxy( factorial, {
   apply: function( target, thisValue, args ) {
        console.log( 'I am called with', args );
        return target( ...args );
   }
});

console.log('Factorial: '+factorial( 5 ));
```



We added a proxy that traps all the calls of the `factorial` method, including the recursive calls. We equate the proxy reference to the `factorial` reference so that all recursive function calls are proxied. The reference to the original factorial function is accessible via the `target` argument inside the proxy.

We will now make two small modifications to the code.

First of all, we will replace

```
return target( ...args );
```

with

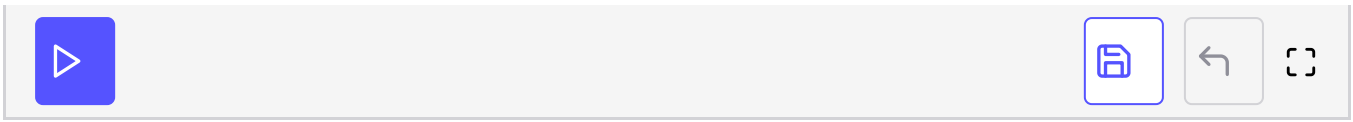
```
return Reflect.apply(
    target,
    thisValue,
    args );
```

We can use the Reflect API inside proxies. There is no real reason for the replacement other than demonstrating the usage of the Reflect API.

Second, instead of logging, we will now count the number of function calls in the `numOfCalls` variable.

If you are executing this code in your developer tools, make sure you reload your page, because the existing code may interact with this

```
let factorial = n =>
    n <= 1 ? n : n * factorial( n - 1 );

let numOfCalls = 0;
factorial = new Proxy( factorial, {
   apply: function( target, thisValue, args ) {
        numOfCalls += 1;
        return Reflect.apply(
            target,
            thisValue,
            args
        );
   }
});

console.log(factorial( 5 ) && numOfCalls);
//> 5
```



In the next lesson, we will learn about controlling proxies and revoking them.