Stack implementation example. Locking things down.

Let's start with a simple stack implementation. We will then consider ways in which someone could mess with the our stack, and will try to protect ourselves against it.

```
// stack.js
export default class Stack {
    constructor() {
        this.values = [];
    }
    push(el) {
        this.values.push(el);
    }
    pop() {
        if (this.isEmpty()) {
            throw new Error("Popping from empty stack");
        } else {
            return this.values.pop();
        }
    }
    isEmpty() {
        return this.values.length == 0;
    }
}
```

And here is a sample use:

```
// main.js
import Stack from './stack.js';
import evil from './evil.js';
let s = new Stack();
evil(s);
s.push(2); s.push(5);
console.log(s.pop());    // 5
console.log(s.pop());    // 2
```

This was not a bad implementation, but it had some "flaws". It allows people to change it. We will try to "lock it down". This is not always a good idea, so think carefully before trying to do this.

Question: Think of the above implementation. You are a nefarious hacker who can run some code after that implementation is completed. You cannot change the value of the Stack variable itself, but you will be able to access the value of the Stack variable. Think of all the ways you could use to affect the behavior of the stack. Some different kinds of attacks to consider:

- Reading or changing parts of a specific stack object that you are not supposed to access.
- Altering the behavior of existing stack objects.
- Altering the behavior of future stack objects.
- Attacking the class Stack or individual stack objects.

So imagine that you can do whatever you want within the file evil.js in the following HTML page:

```
<!doctype html>
<html>
<head>
   <title></title>
</head>
<body>
   <script type="module" src="stack.js"></script>
   <script type="module" src="evil.js"></script>
   <script type="module" src="main.js"></script>
</body>
</html>
```

Think through various attack scenarios to disrupt the behavior of main.js, and how we might change stack.js to protect against them.

We are going to go through some of the vulnerabilities below, but try to think of some yourselves first.

Weak prototype

As it stands, the stack prototype is accessible and can be changed. A first attempt would be the following:

```
// evil.js
Stack.prototype = {};
```

Luckily for us, part of using the class construction is that the prototype property is set to not be changeable:

```
console.log(Object.getOwnPropertyDescriptor(Stack, "prototype"));
// Returns:     writable: false, enumerable: false, configurable: false
```

However, we could do the following:

```
// evil.js
import Stack from './stack.js';
export default function evil(s) {}
Stack.prototype.push = function(el) {};    // Now pushing to the stack does nothing.
```

We can guard against this by "freezing" the stack prototype:

```
// stack.js
...
Object.freeze(Stack.prototype);
```

Here's the MDN documentation on Object.freeze[1]. It says:

> The Object.freeze() method freezes an object. A frozen object can no longer be changed; freezing an object prevents new properties from being added to it, existing properties from being removed, prevents changing the enumerability, configurability, or writability of existing properties, and prevents the

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze

values of existing properties from being changed. In addition, freezing an object also prevents its prototype from being changed. freeze() returns the same object that was passed in.

So this will prevent someone from messing with the prototype methods.

This of course has consequences. It will likely be more difficult for you to create spies for testing.

Weak values

The values array which stores the stack values is currently accessible by anyone with access to the stack object. For instance, our evil function can do the following:

```
// evil.js
export default function evil(s) {
   delete s.values;
}
```

We can fix this, by freezing the object we create in the constructor:

```
export default class Stack {
    constructor() {
        this.values = [];
        Object.freeze(this);
    }
    ...
```

Note that this has consequences: noone, not even you, can change the values property any more.

Visible values

The mere fact that the values property is visible is somewhat of a vulnerability. Someone can see the values stored in our stack. We can try to protect against it, but it's not easy. We can use Object.defineProperty to make the values property not enumerable (and not configurable), but this is only a temporary solution as the variable is still accessible to those that know its name. We need to somehow *hide* the name. But how can we do this while still being able to access it?

The trick is to generate the name when the Stack module is first loaded, then use this name instead of "values". This could be as simple as appending a randomly generated number to the word "values" like so:

```
const key = "values" + Math.random();
export default class Stack {
    constructor() {
        this[key] = [];
        Object.freeze(this);
    }
    push(el) {
        this[key].push(el);
```

```
    }
    pop() {
        if (this.isEmpty()) {
            throw new Error("Popping from empty stack");
        } else {
            return this[key].pop();
        }
    }
    isEmpty() {
        return this[key].length == 0;
    }
}

Object.freeze(Stack.prototype);
```

If you were to now add a console.log(s) to your main.js, you would see something like:

```
Stack {values0.4841189913080053: Array(0)}
```

And more importantly, this number will be different each time your page loads. And this number is stored in the constant key which is private to the stack.js module, so noone else can get access to it.

Now we can hide this property with our usual Object.defineProperty trick:

```
// stack.js
export default class Stack {
    constructor() {
        Object.defineProperty(this, key, {
            value: [], writable: false,
            enumerable: false, configurable: false
        });
        Object.freeze(this);
    }
    ...
```

Alas, this doesn't quite work. Someone can still discover the property name as follows:

```
// evil.js
import Stack from './stack.js';
export default function evil(s) {
   let descriptors = Object.getOwnPropertyDescriptors(s);
   console.log(Object.keys(descriptors)); // shows the values... property
   let prop = Object.keys(descriptors)[0];
   console.log(s[prop]);      // the values array!
}
```

The only way to protect ourselves from this is to make the property somehow a "local" variable of the created object. But this means that each object should have its own pop and push and isEmpty methods, that have access to this distinct local variable. This means giving up one of the benefits of using objects in the first place, but it is an option if it comes to it.

Array implementation

The values property, even after its true name is hidden, still holds an array. This means that someone can still mess with it by messing with the Array class:

```
/// evil.js
Array.prototype.push = function() {};
```

Now our this[key].push(el); method will call this altered array method, and will therefore not add anything to our array. We can similarly mess with the pop method instead:

```
/// evil.js
Array.prototype.pop = function() { return  1; };
```

To prevent this from happening, we could try to freeze the Array prototype:

```
Object.freeze(Array.prototype);
```

But one might view this as a nuclear option:

```
Many library add their own functionality to the ‘Array.prototype‘ object, to either fill in n
```

Bottom line is that you are now messing with code that does not belong to you, and that always comes at a price.

What other options do we have? We could use a linked list instead. But a simpler solution might be to simply make copies of the properties we want to use from the Array class. Then we will use those copies when we need to work with the array, and we will call them via call:

```
const key = "values" + Math.random();
const push = Array.prototype.push;
const pop = Array.prototype.pop;
export default class Stack {
    ...
    push(el) {
        push.call(this[key], el);
    }
    ... // similar for pop
```

Now if someone messes with the Array.prototype.push and its friends, we don’t care because we’ve kept our own local copies.

Did you notice the one thing we left out however? We are calling the Function.prototype.call method. Someone could still mess with that:

```
// evil.js
Function.prototype.call = function() { }; // Makes ".call" do nothing
```

How can we protect against that? We could create a local copy of call and try to use it, but how would we call that copy so that it has the correct this object (namely the function to be executed)? The following doesn’t work:

```
// stack.js
const _call = Function.prototype.call;
...
    push(el) {
```

```
        _call(push, this[key], el);
    }
...
```

This does not work, because _call is now called as a function, and therefore does not have the correct this object for it to do its work.

We can fix this, in a pretty crazy way, by binding the this to this call function to be itself:

```
const _call = Function.prototype.call.bind(Function.prototype.call);
```

Phew, that was some tricky stuff!

But wait, there's more! We also used Object.defineProperty in the constructor, to hide the object. That can also be abused by an evil code:

```
// evil.js
Object.defineProperty = function() {};
// Could also do something smarter to learn the values property name
// then make it editable or whatnot.
```

So to protect against that, we would need our own copy of that as well (and to call it appropriately):

```
// stack.js
const dp = Object.defineProperty;
export default class Stack {
    constructor() {
        _call(dp, Object, this, key, {
            value: [], writable: false,
            enumerable: false, configurable: false
        });
        Object.freeze(this);
    }
    ...
```

Oooh did you notice the Object.freeze there? That needs protecting as well!

```
// evil.js
import Stack from './stack.js';
export default function evil(s) {
   // Find out values name by earlier trick
   s[valuesName] = []; // Put your own thing here, maybe something that looks like an array
}
Object.freeze = function(o) { return o; };
```

And to fix it:

```
// stack.js
const fr = Object.freeze;
        ...
        ...
        _call(fr, Object, this);
    }
```

Cool, eh?

By the way, all of these will only protect you from evil code that runs *after* your module is loaded. It will not protect you from code that messes up with these fundamental functions *before* your module is loaded.