Cheat Sheet – Proxy API

Metaprogramming

Metaprogramming means that you're able to change (parts of) the behavior of the underlying language – JavaScript in this case. This of course is a powerful feature as it allows you to influence the way your code is executed. The Reflect API (like Symbols and Proxies) are important additions which help you with Metaprogramming – something that wasn't really possible in JavaScript before.

What it Does

The Proxy API allows you to wrap objects, functions, whatever and trap/handle incoming property accessing, function calls etc. You may think of Proxies as filter or barrier which has to be passed and which may interrupt access on the wrapped element.
For example you might wrap a Proxy around an object and set up a trap (that's what these functions are called) to be triggered whenever something (the source code) tries to access a property of the wrapped object. The Proxy can then interrupt this access and maybe deny it, return another value, run some calculation – whatever you want.

Wrapping Objects

An object could be wrapped like this:

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
let person = new Person('Max', 27);
let proxy = new Proxy(person, {
    // Setup traps here
    get: function(target, property, receiver) {
        return 'Something else';
    }
});
```

Notice that here, a get trap is set up – triggered whenever something tries to get the value of a property.
The function triggered in this case passes the target of the access, the accessed property and the receiver of the value as an argument. Inside the trap you may do whatever you want to do.

Wrapping Functions

You may not only wrap objects. Functions can be wrapped too:

```
function log(message) {
    console.log('Log entry created: ' + message);
}
let proxy = new Proxy(log, {
    apply: function(target, thisArg, argumentsList) {
        if (argumentsList[0].length < 20) {
            return Reflect.apply(target, thisArg, argumentsList);
        }
        return false;
    }
});
proxy('Hello!');
proxy('Hello, this is a very long message!');
```

In this example, the second function call would fail since the message is too long.

Proxies as Prototypes

You can also use Proxies as prototypes:

```
let person = {
    name: 'Anna'
};
let proxy = new Proxy({}, { // notice the empty object!
   get: function (target, property, receiver) {
       return 'Property ' + property + ' not found!';
   }
});
Reflect.setPrototypeOf(person, proxy);
console.log(person.name);
console.log(person.age); // not found
```

Revocable Proxies

A special case are revocable proxies. Unlike “normal“ proxies, those proxies (created via Proxy.revocable(), without *new* keyword!) can be revoked. Not very surprising, considering the name.

Available Traps

More information on Proxies as well as a list of all traps can be found here: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Proxy