JS Closures

Because closures are a difficult notion. . .

Overview

- constructing privates
- defining closures
- constructing modules
- classic closure mistakes

pdf

JS Object

A JS object is the equivalent of a Java object with all fields declared as public

```
let a = {v1: 10, v2: "oqroi"};

console.log(a.v1);
> 10

a.v1 += 1;

console.log(a.v1);
> 11
```

Missing : private keyword

A function as property of an object

```
let a = {v1: 10, v2: "oqroi"};

a.f = function(x) { return x+1;}

a.f(4)
> 5
```

this

A function which is a property of an object has the keyword `this` set to the object.

```
let a = {v1: 10, v2: "oqroi"};

a.g = function() { return this.v1 = this.v1+1;}

a.g()
> 11
```

Or written in another way :

```
let a = {v1: 10, v2: "oqroi", g: function() { return this.v1 =

a.g()
> 11
a.g()
> 12
```

Variable scope

f1, variable inside of f, is not visible outside

```
function f() {
  let f1 = 4;
  console.log(f1);
}

f()
> 4
f1
> ReferenceError
```

Creating a private scope

```
function f() {
  let f1 = 4;
  return function() { console.log(f1); }
}

let g = f() // f() returns a function referring to the variabl

f1
> ReferenceError
g()
> 4 // f1 still exists
```

By creating a function inside another function :

- I can create a way to access a variable that is not accessible in any other way : a `private`
- the variable is kept as long as the access function exists

This private is using a closure

The closure is constructed with :

- a function outside
- some variables in the function
- a function defined inside and referring to variables of the outside function

The closure "is" the set of variables known inside the outside function.

The closure exists as long as the inside function exists == can be accessed.

As soon as the inside function is not referred to by any variable, the function and the closure are garbage-collected.

Same as before with anonymous functions

```
let g = (function () {
  let f1 = 4;
  return function() { console.log(f1); }
})(); // the last two parentheses are the call to the anonymou

f1
> ReferenceError
g()
> 4 // f1 still exists
```

Creating more access functions

```
let h = (function () {
  let f1 = 4; // private
  let i = {};
  i.get = function() { return f1; } // getter
  i.set = function(x) { f1 = x; } // setter
  return i;
})(); // the last two parentheses are the call to the anonymou

h.get()
> 4
h.set(10)
> undefined
h.get()
> 10
```

What is missing ?

So we have ways to :

- define private variable
- define getters
- define setters
- return as many functions from a private scope == export

Missing : a way to import stuff

../tp ../logo-IPP-s

Importing

```
const importer = require("someModuleName");

let h = (function (imp) {
  let f1 = imp.foo.bar;
  let i = {};
  i.get = function() { return f1; } // getter
  i.set = function(x) { f1 = x; } // setter
  return i;
})(importer);
```

With this, we have a way to import another set, and rename it if necessary.

To conclude, we have a full `module` system.

Two module syntaxes

There are two frequent module syntaxes :

- CommonJS (the old one)
- ES6 (the new one)

Module syntax : CommonJS

A JS file can be considered as a module, if it finishes on something like :

```
// at the end of toto.js
exports.fibonacciIt = fibonacciIt;
exports.fibonacciRec = fibo_rec;
exports.fibonacciArray = fibonaArr;
exports.fibonacciMap = fibonacciMap;
```

In that case, in another file, you can use :

```
let totoMod = require("toto");

console.log(totoMod.fibonacciRec(4));
```

If the file extension is `.cjs`, node understands this file to be in CommonJS syntax. If the file extension is `.js` and the `package.json` does not have a `"type": "module"`line, node also

Module syntax : ES6

- To import fun from module npmmod (managed by npm), use : `import {fun} from 'npmmod';`
- To import fun from module mod (yours, residing in mod.js or mod.mjs), use : `import {fun} from './mod';`
- You can import more : `import {fun, fun2, fun1 as foo} from 'mod';` where I renamed fun1 as foo for use in the current file
- If you want to manipulate the module itself : `import default from 'mod'` then you can use `mod.fun`, `mod.fun2`, etc.
- You can rename the module with : `import default as othername from 'mod'`
- You can export a const, a let or a function by putting `export` in front of the definition

If the file extension is `.mjs`, node understands this file to be in ES6 syntax. If the file extension is `.js` and the `package.json` has a `"type": "module"`line, node also understands this file to be in

Modules

- there are multiple module systems
- the best known module manager for node.js is npm
- there are many module systems for the browser
  - require.js : just deals with importing all the dependencies in the right order from one line of HTML
  - webpack : import all dependencies, compile single bundle with many options,. . .
  - browserify, bower, gulp, grunt. . .

More on closures

A classic mistake

```
function f() {
    for (var i=0; i<3; i++) {
      setTimeout(function(){ console.log(i);}, 1000);
    }
}

f();

// 1s later
3
3
3
```

Not a fix

This does not work

```
function f() {
    for (var i=0; i<3; i++) {
      setTimeout(function(){var j = i; console.log(j);}, 1000
    }
}

f();

// 1s later
3
3
3
```

There are three variables named j, but they are set later from a reference to the same i.

../tp ../logo-IPP-s

Closures : Fix

```
function g(j) {
    return function() { console.log(j); };
}

function f() {
    for (i=0; i<3; i++) {
      setTimeout(g(i), 1000);
    }
}

f();
0
1
2
```

- Solution :
  - Problem is solved by calling another function n times
  - The value of i is copied during the loop, not after (when g is

Closures : Fix 2

```
function f() {
    for (i=0; i<3; i++) {
      setTimeout((function (j) {console.log(j);})(i), 1000);
    }
}

f();
0
1
2
```

- More compact : create an anonymous function called immediately
    - current practice in JS libraries
    - not recommended : less readable

Summary of this lesson

- object with a function as property, this, variable scope
- creating private and getter, closure
- access functions, export, import, modules
- typical closure mistake