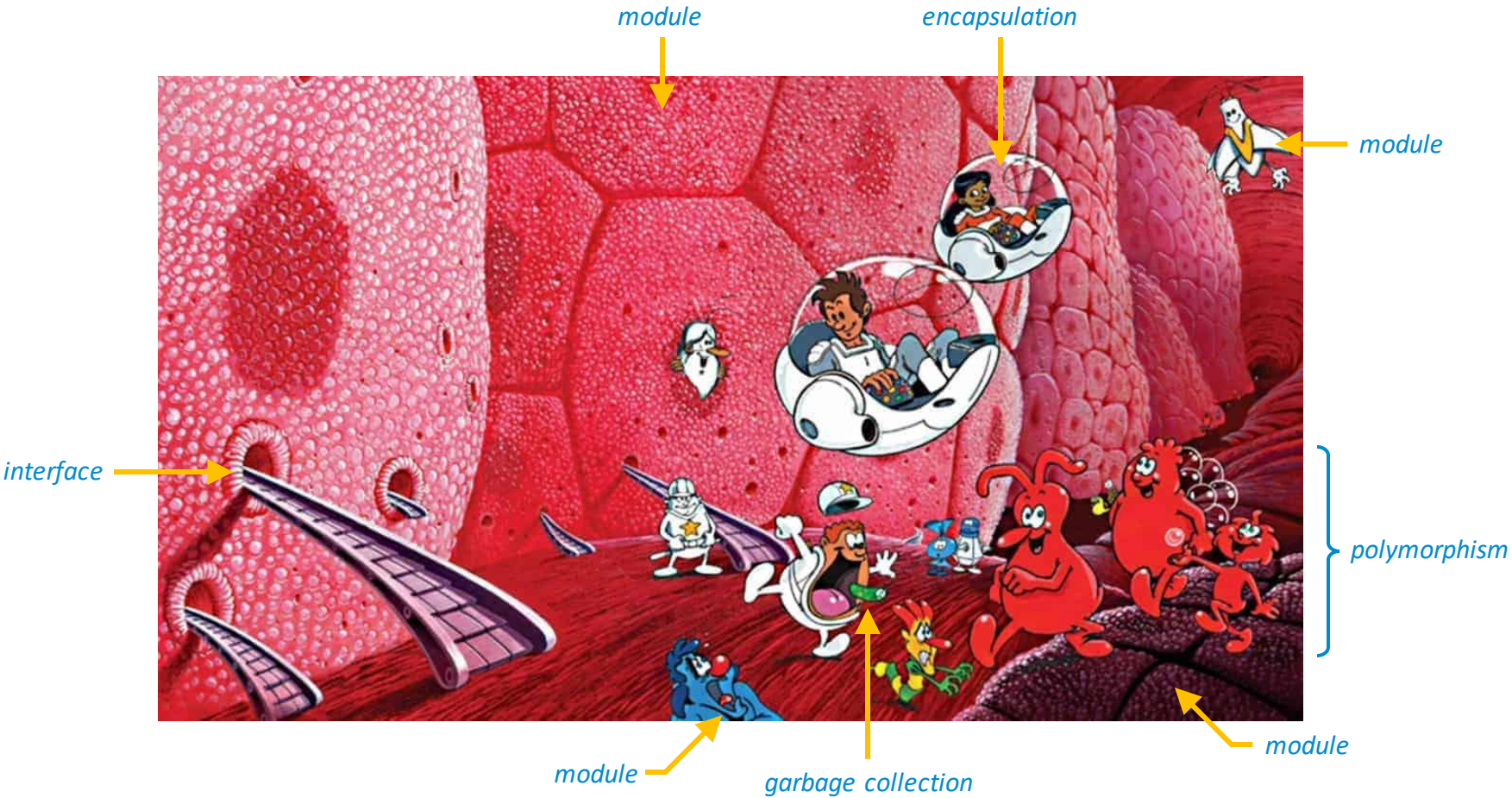
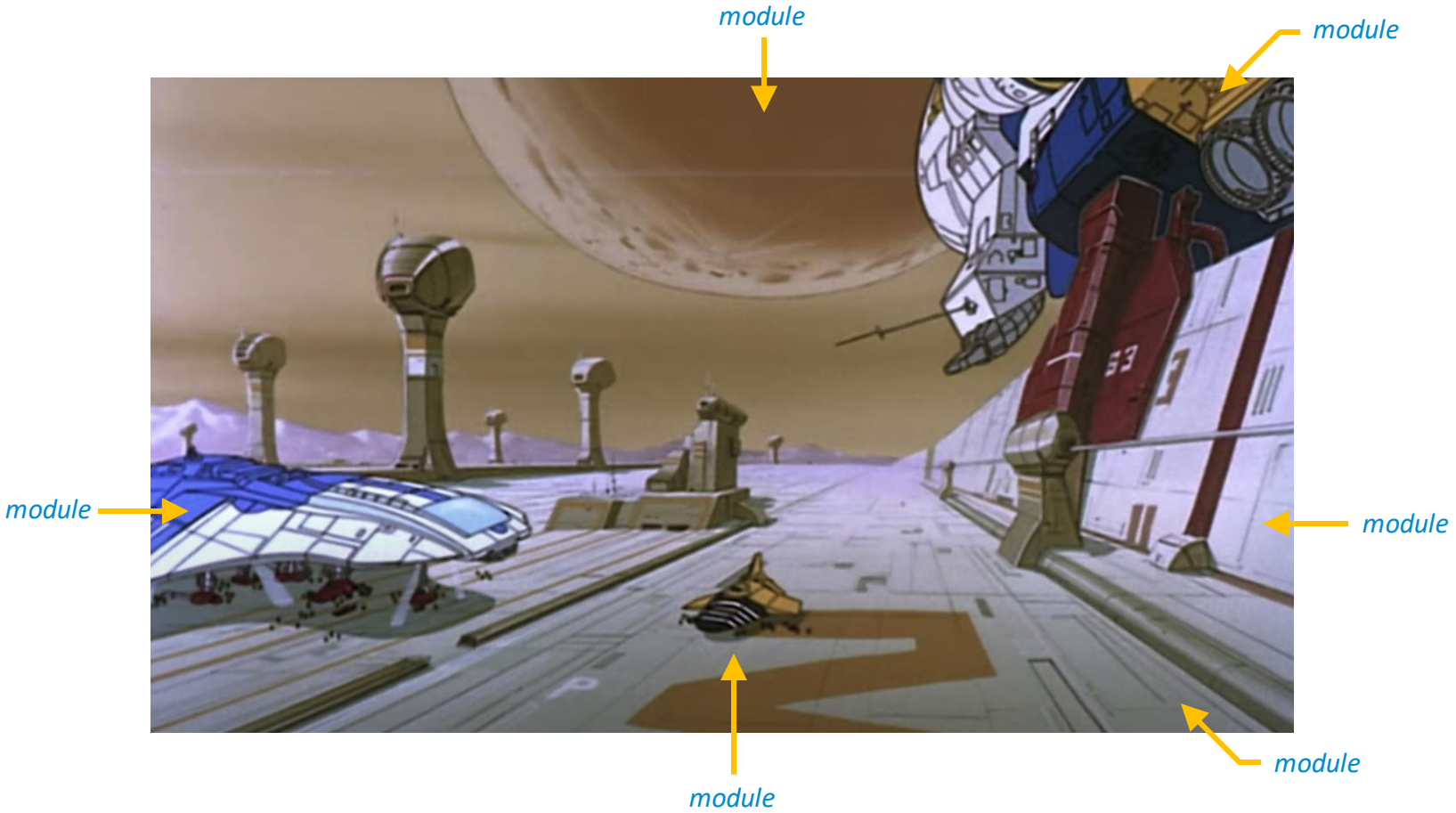
~~CommonJS, AMD~~

ECMAScript Modules

Frontend Junior Program - 2022


module
encapsulation
module
interface
polymorphism
module
garbage collection
module


module
module
module
module
module
decomposition to smaller modules


module
module
module
module
module
module

Everything is **module**

Things consist of parts

And program code is not different. As you write code, you will naturally break it down into smaller, reusable components.

The question is not why, rather than: *how to do it properly?*

Every code, in every programming language can be modularized.

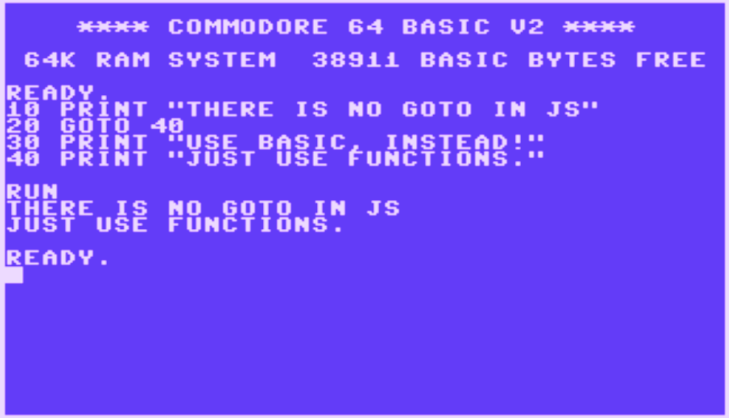
```
/*---------------------------------------------------------------
* Program  : Subroutine for that multiplies two 8-bit signed
* Input    : The input parameters are:
             R12 -- array starting address
             R13 -- the number of elements (assume it is =>1
             R14 -- dispay (0 for P1&P2 and 1 for P3&P4)
* Output   : No output parameters
*---------------------------------------------------------------
#include "msp430.h"                    ; #define controlled

        PUBLIC suma_rp

        RSEG CODE

suma_rp:
        ; save the registers on the stack
        PUSH    R7                     ; temporal sum
        CLR     R7
lnext:  ADD     @R12+, R7              <- R12: input parameter
        DEC     R13
        JNZ     lnext                  <- JNZ: jump if not zero = if
        BIT     #1, R14                ; display on P1&P2
        JNZ     lp34                   ; it's P3&P4
        MOV.B   R7, P1OUT
        SWPB    R7
        MOV.B   R7, P2OUT
        JMP     lend                   <- JMP (JUMP) = goto
lp34:   MOV.B   R7, P3OUT
        SWPB    R7
        MOV.B   R7, P4OUT
lend:   POP     R7                     ; restore R7
        RET                            <- RET = return
        END
```

Even assembly programs are modularized to subroutines (functions)

Goto

There is no *goto* in JavaScript

We have *break* and *continue*, but those are different. In JavaScript, the smaller meaningful module can be the *function*.



A properly modularized code using GOTO

Dependency

Modules are built on top of others

The modules' hierarchy is like a tree: top level modules depends on lower-level ones.

the *add* function is an abstraction over the "+" operator;
(we can use it to add anything, not just numbers; this way, we encapsulated the logic of the addition; later we can extend the functionality by changing the internals, and we have to change only here, there is no "+" anywhere else)

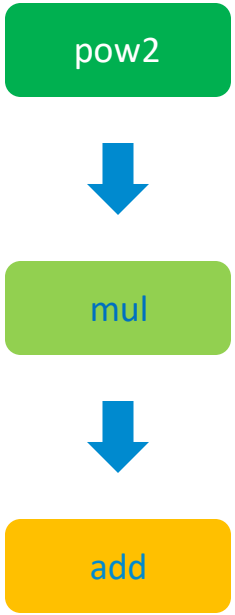
mul is an abstract multiplication function;
(please note that we can multiply anything: e.g., vectors, matrices, etc., the *mul* does not know about the type of its multiplicand (n))

add is dependency of *mul*

```
function add(a, b) {
  return a + b;
}

function mul(n, multiplier) {
  return Array(multiplier).fill(n).reduce((acc, e) => add(acc, e));
}

function pow2(n) {
  return mul(n, n);
}

mul(3, 4);
12
pow2(5);
25
```

mul is a dependency of *pow2*

Dependency graph



pow2() depends on *mul()*

mul() depends on *add()*

transitive dependency:
pow2() depends on *add()*

it is simple, right? let's see a bit more trouble then…

Circular dependency

clicking on *Help* will really render the Help page – this is a fully functional SPA, with a router.

Please choose an action! >> *File* :: *Edit* :: *Help*

Modules can depend on each other

The circular dependency is very common, and it can cause several issues, such as unintended recursion calls.

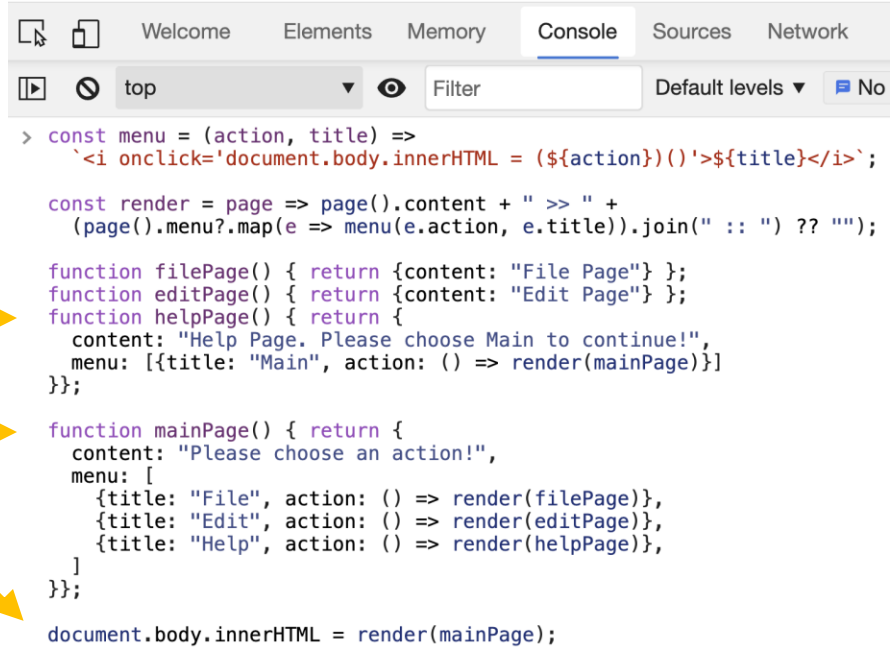
Also, while this code may look properly modularized, it is not, as *mainPage* and *helpPage* are *tightly coupled*.

Usually, you'll face with this problem, when you try to break down this code into separate modules.

helpPage depends on *mainPage*

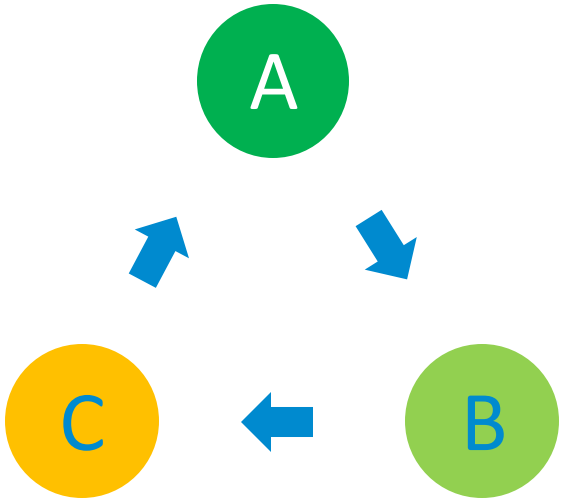
mainPage depends on *helpPage*

because of the hoisted function declarations, the circular dependency is not an issue (there is no error)



Circular dependency, however, is a thing

So, it is expected from a module system to handle it properly. Still, you should avoid from it.



A lecture spin-off

Just stop for a moment

This is how things can go wrong quickly. I hope that you already spotted some nasty code parts in our awesome SPA.

innerHTML in an *innerHTML*, (menu in render in the assignment: nice! *not*.

it relies on the a global variable (render), which could be problematic later

as mentioned, function declarations can refer to each other – it is pretty easy to create a very much tangled code this way

it could be a good idea to order the functions in the code in dependency order

innerHTML is almost always a no go; it is basically an *eval* – and we just don't know whether the *render* and *mainPage* are safe enough

```
> const menu = (action, title) =>
    `<i onclick='document.body.innerHTML = (${action})()'>${title}</i>`;

  const render = page => page().content + " >> " +
    (page().menu?.map(e => menu(e.action, e.title)).join(" :: ") ?? "");

  function filePage() { return {content: "File Page"} };
  function editPage() { return {content: "Edit Page"} };
  function helpPage() { return {
    content: "Help Page. Please choose Main to continue!",
    menu: [{title: "Main", action: () => render(mainPage)}]
  }};

  function mainPage() { return {
    content: "Please choose an action!",
    menu: [
      {title: "File", action: () => render(filePage)},
      {title: "Edit", action: () => render(editPage)},
      {title: "Help", action: () => render(helpPage)},
    ]
  }};

  document.body.innerHTML = render(mainPage);
```

File based modules

Breaking down into files

While modularity can be achieved within one file, it does make sense to break down the code into separate files.

but breaking down into modules without reasons could be problematic as well

- it is easier handle smaller files (overview, navigate, edit)
- the tree structure of the directory system provides a visually clear hierarchy between modules

if used properly; many times it is the first sign of the structural problems if files cannot be placed into a tree hierarchy

but modularization won't protect from creating a tangled mess;

in fact, it is easier to do, because the problems are less visible and could remain under the radar for a prolonged time

- concerns are separated, harder do write spaghetti code
- interfaces and dependencies are clearly visible
- team members can work independently on different code parts

many times teams can modify only their modules; even within a team it could be forbidden to edit common files during feature development

JavaScript ES5

No modules

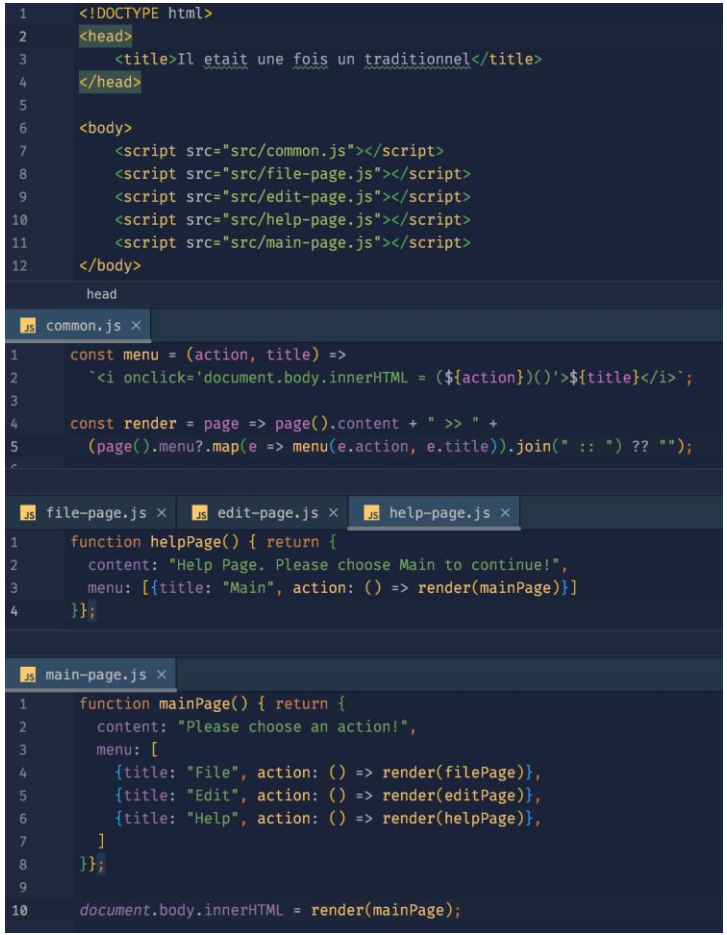
Until ES 2015, there were no module system in JavaScript.

While it is possible to split the code into separate files, all the code shares the same global namespace: every variable in any file is accessible in other files as well.

Files must be referred in html in dependency order – if you have 35 JS files to load (even when those are bundled together) this could be very error prone.

Also, the interfaces are not clear: what is the input and what is the output? Does a file ("module") use global variables?

A simple solution could be the Module pattern.



```html
<!DOCTYPE html>
<head>
    <title>Il etait une fois un traditionnel</title>
</head>

<body>
    <script src="src/common.js"></script>
    <script src="src/file-page.js"></script>
    <script src="src/edit-page.js"></script>
    <script src="src/help-page.js"></script>
    <script src="src/main-page.js"></script>
</body>
```

head

common.js

```
const menu = (action, title) =>
  `<i onclick='document.body.innerHTML = (${action})()'>${title}</i>`;

const render = page => page().content + " >> " +
  (page().menu?.map(e => menu(e.action, e.title)).join(" :: ") ?? "");
```

file-page.js | edit-page.js | help-page.js

```
function helpPage() { return {
  content: "Help Page. Please choose Main to continue!",
  menu: [{title: "Main", action: () => render(mainPage)}]
}};
```

main-page.js

```
function mainPage() { return {
  content: "Please choose an action!",
  menu: [
    {title: "File", action: () => render(filePage)},
    {title: "Edit", action: () => render(editPage)},
    {title: "Help", action: () => render(helpPage)},
  ]
}};

document.body.innerHTML = render(mainPage);
```

Module pattern

the issue with the circular dependency is clearly a problem now (helpPage tries to use mainPage, but that will be defined later), however, it is nothing, that an ingenious developer cannot solve with a little ~~hack~~ tactical workaround

This simple solution utilizes the IIFE

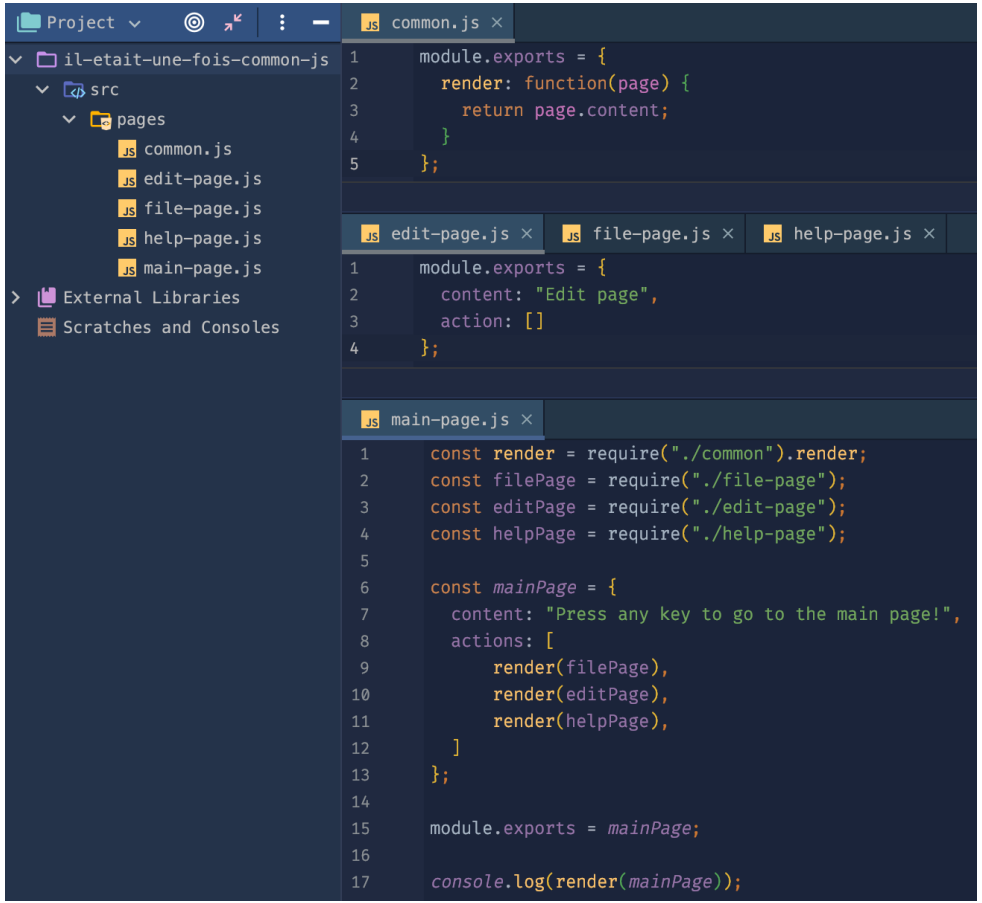
The IIFE* provides a kind of encapsulation.

While it is still possible to use global variables, it encourages using well defined interfaces. Still, the "exported" modules are globally accessible.

Loading of the "modules" still should be done in dependency order. It is not really a module system, it is the same as a global JS, just in a bit safer way.

The Revealing Module Pattern is essentially the same approach.

** Immediately Invoked Function Expression - you should know this at this point.*



CommonJS module system

Node.js, however, utilized a more explicit solution, a module system that is based on CommonJS (while not exactly the same) – it does have real, independent modules. It solves the global namespace issue, as any identifier declaration remains local, so the module must have to export its internals to be accessible.

You will meet this on any project with the JS based configuration files of the tool-chains.

← imports with *require*, the .js extension is not required (sic!)

← exports with the *module* object

CommonJS

at least a warning on the circular dependency

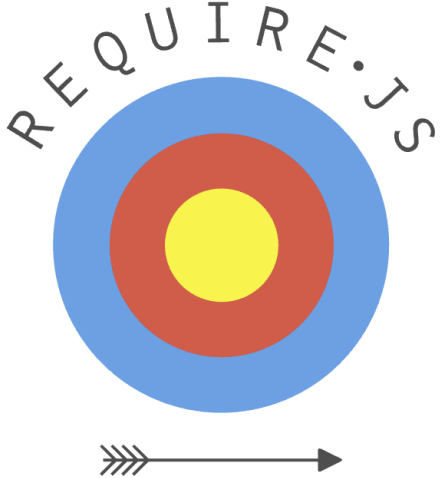
```
~/il-etait-une-fois> node src/pages/main-page.js
Press any key to go to the main page!
(node:97071) Warning: Accessing non-existent property 'content' of module exports inside circular dependency
(Use `node --trace-warnings ...` to show where the warning was created)
```

Asynchronous Module Definition

Before webpack, AMD was the predominantly used module system in the JavaScript ecosystem

Today is not (however, with webpack, you can still use). AMD is basically a new layer above the Module pattern. A thick layer. If you are interested, here is link for your consideration.

The Require.JS implementation is based on AMD.



JavaScript ES6 Modules

From ES 2015, JavaScript provides a unified way - the JavaScript modules

JavaScript modules uses simply *import* and *export* statements, instead of *require* and *module.exports* in CommonJS.

The only caveat here is that the Internet Explorer does not support it, therefore it still requires a build system to create a bundle for IE and pass the standard ES Modules to the proper browsers.

Practically, every project uses a bundler (e.g., for minifying the code), but nowadays the ES Module syntax is preferred (webpack supports it).

How does it work? Let's analyze this!

type “module” is required; scripts are deferred by default

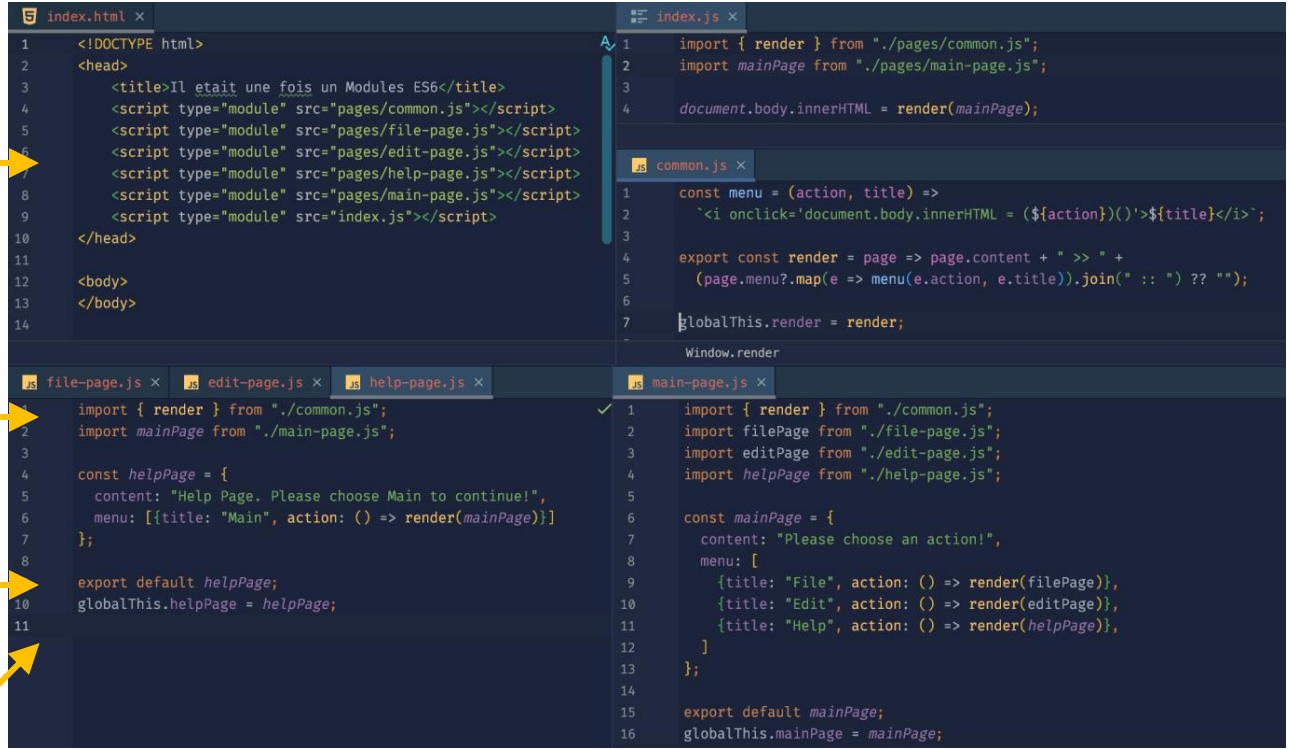
backward compatibility can be provided with the *nomodule* attribute on the script tag

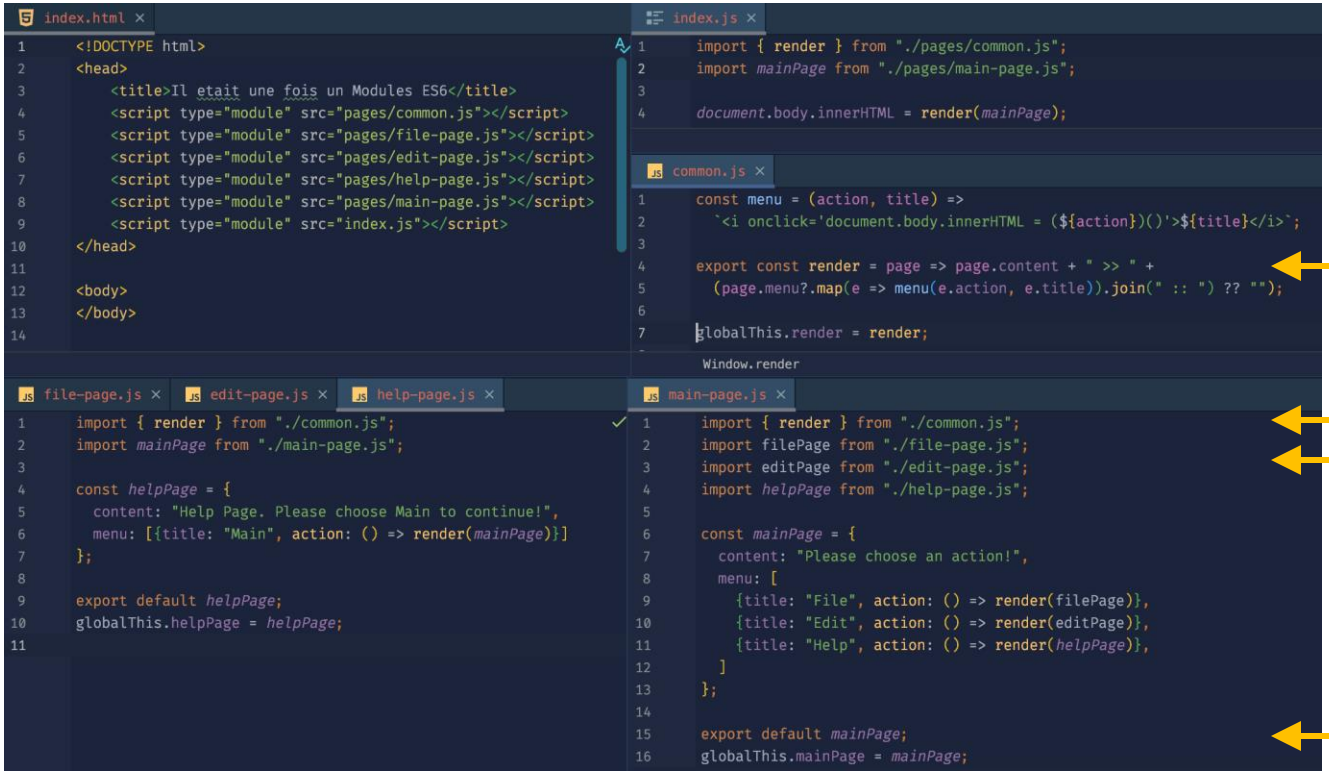
it uses import

and export

variables can be added to the global object with *globalThis*

however, it is almost never ever a good idea to utilize global variables in a module





exports can be named...

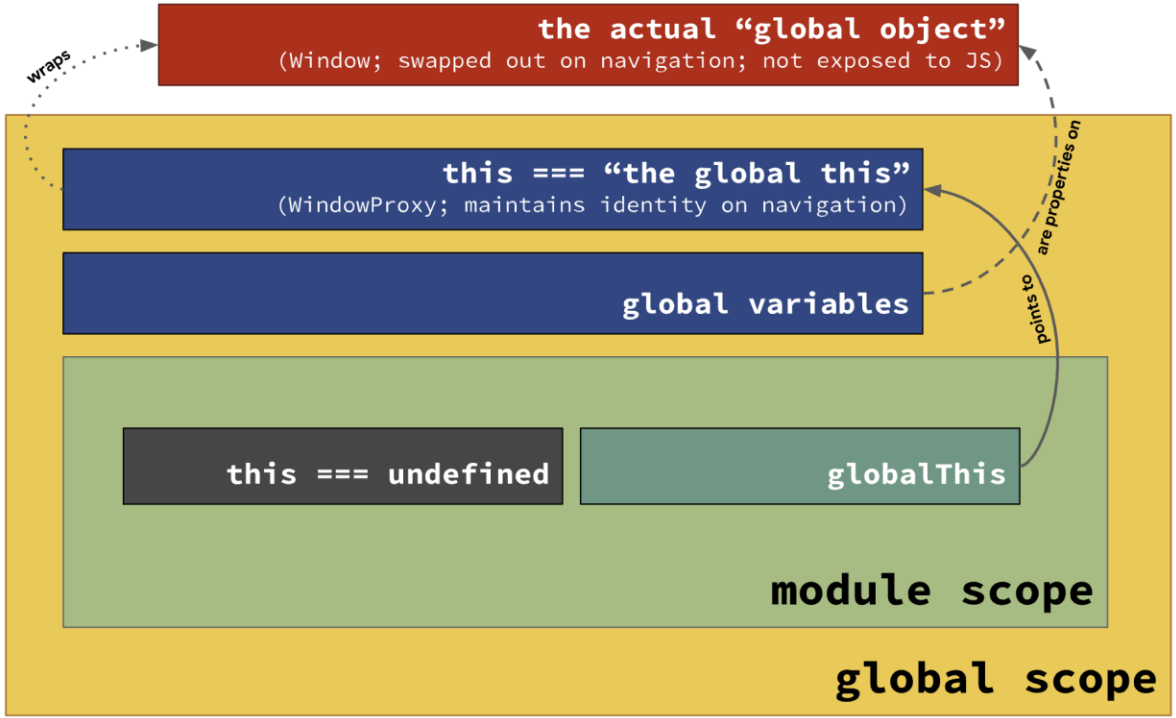
importing named exports...

...and default exports

...or default

edu_hu@epam.com

Appendix



there is an excellent article on why *globalThis* is not a trivial thing