



~~CommonJS, AMD~~

ECMAScript Modules

Frontend Junior Program - 2022

CONFIDENTIAL | © 2022 EPAM Systems, Inc.

Achievement despite ignorance

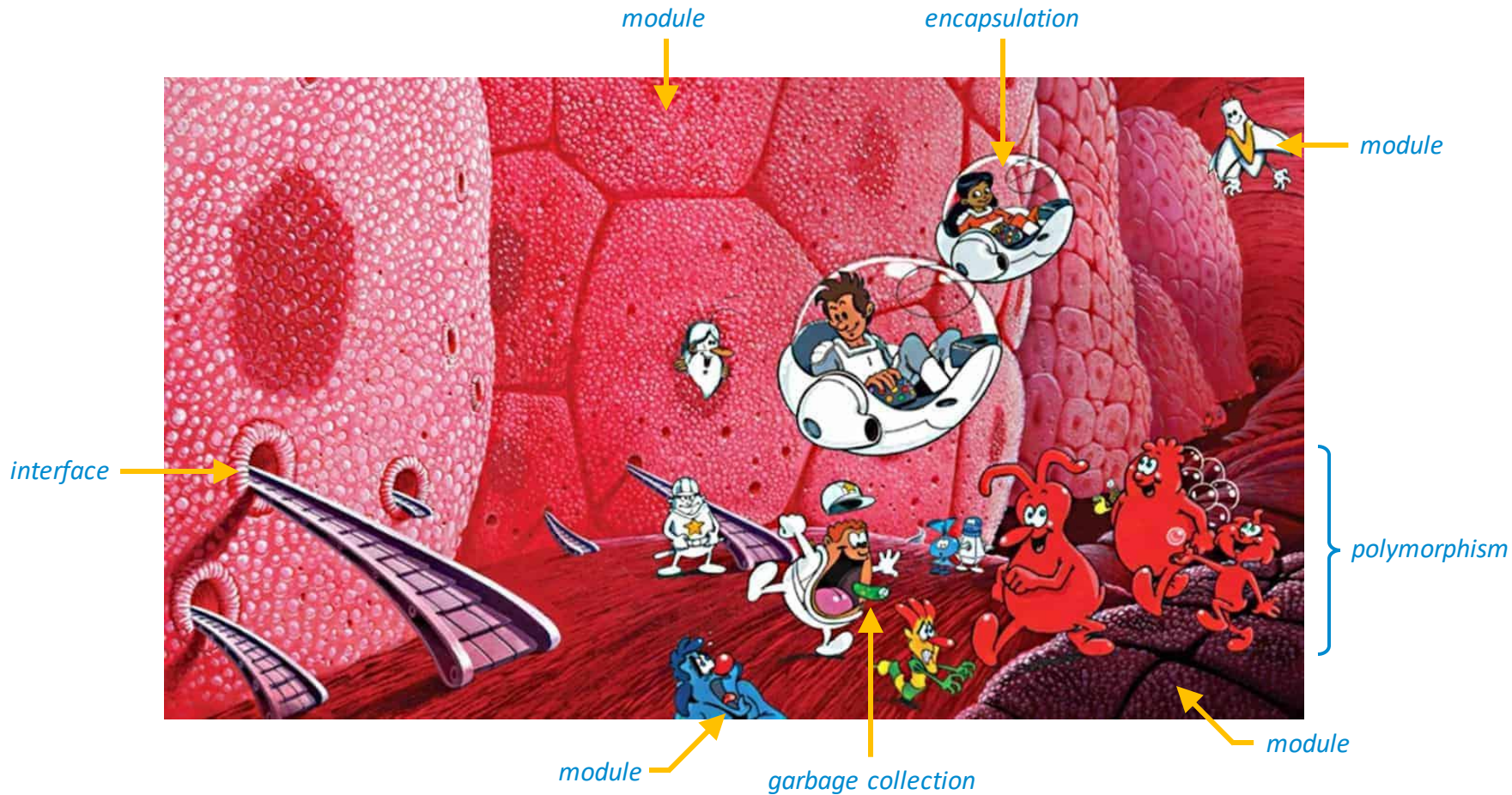


Getting it to
Work

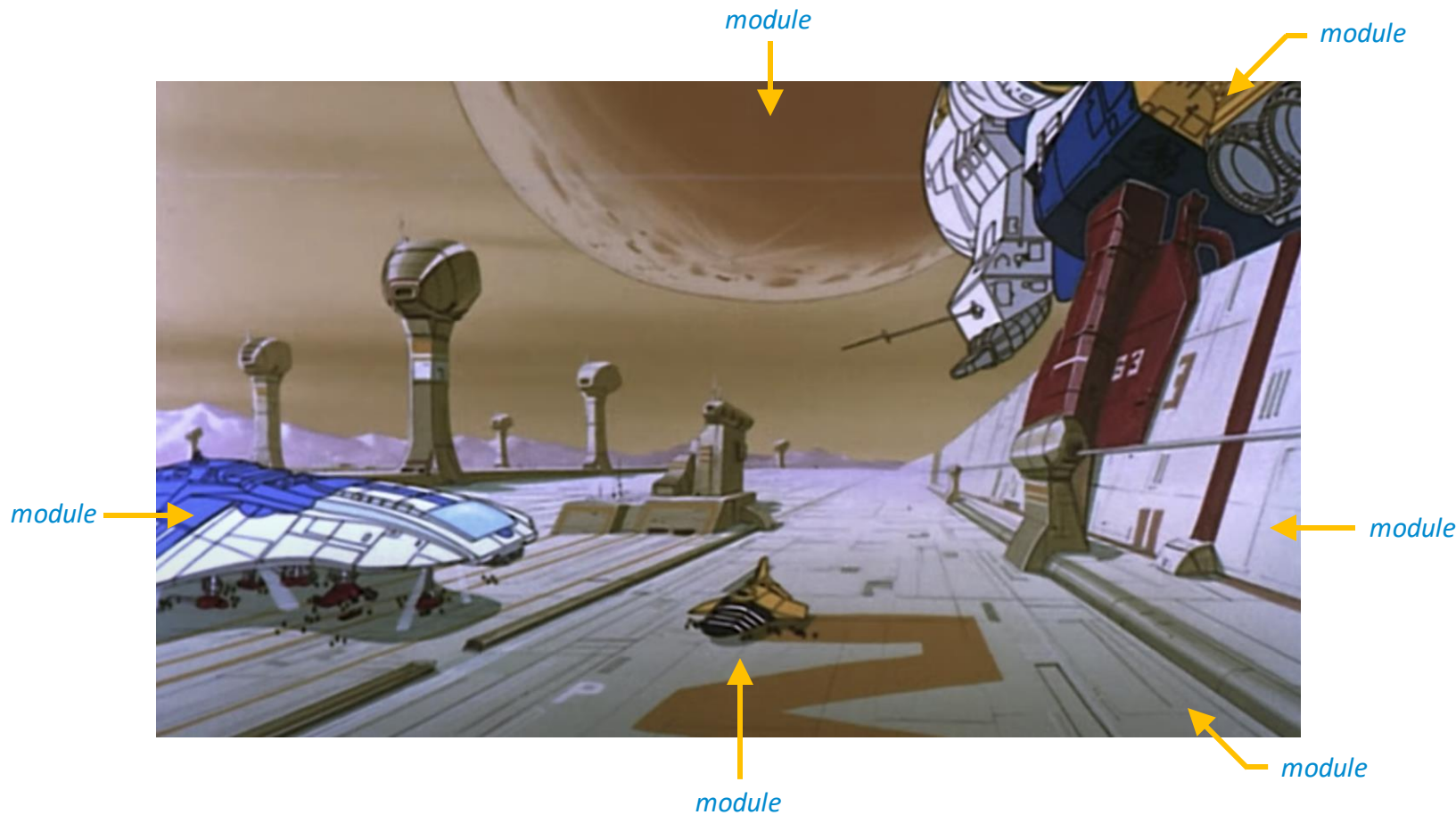
And Having No Idea How

ORLY?

@ThePracticalDev







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 -- display (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*.

```
**** COMMODORE 64 BASIC V2 ****
64K RAM SYSTEM  38911 BASIC BYTES FREE
READY.
10 PRINT "THERE IS NO GOTO IN JS"
20 GOTO 40
30 PRINT "USE BASIC, INSTEAD!"
40 PRINT "JUST USE FUNCTIONS."

RUN
THERE IS NO GOTO IN JS
JUST USE FUNCTIONS.
READY.
```

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))

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

add is dependency of **mul**

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

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

mul is a dependency of **pow2**

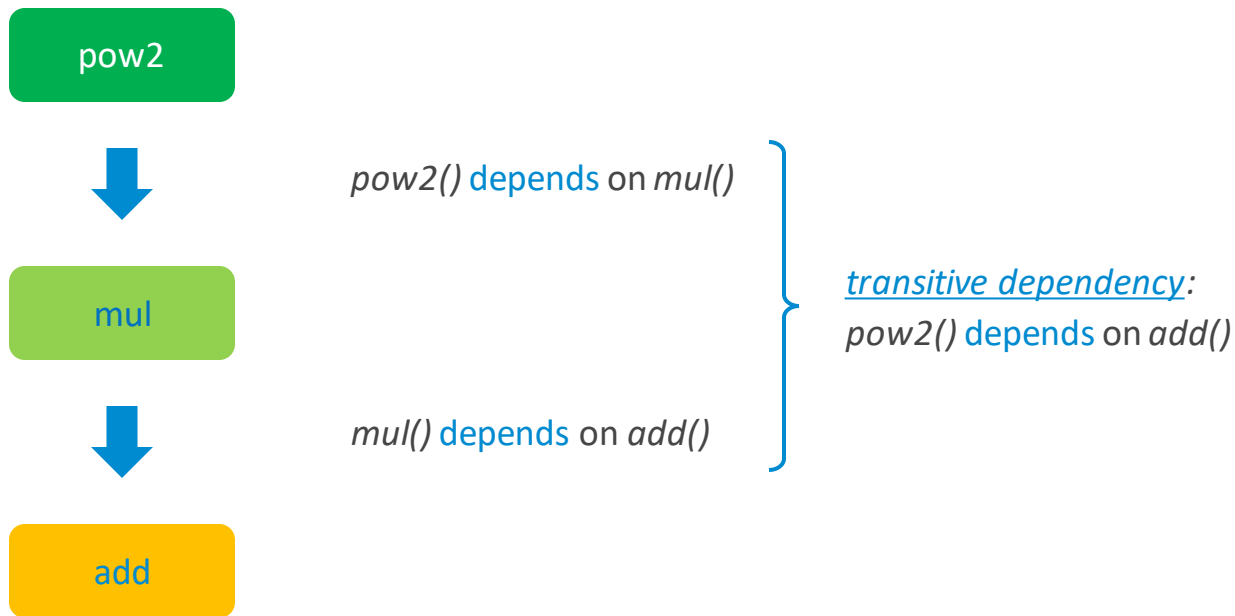
```
mul(3, 4);
```

```
< 12
```

```
> pow2(5);
```

```
< 25
```

Dependency graph



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

Circular dependency

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.

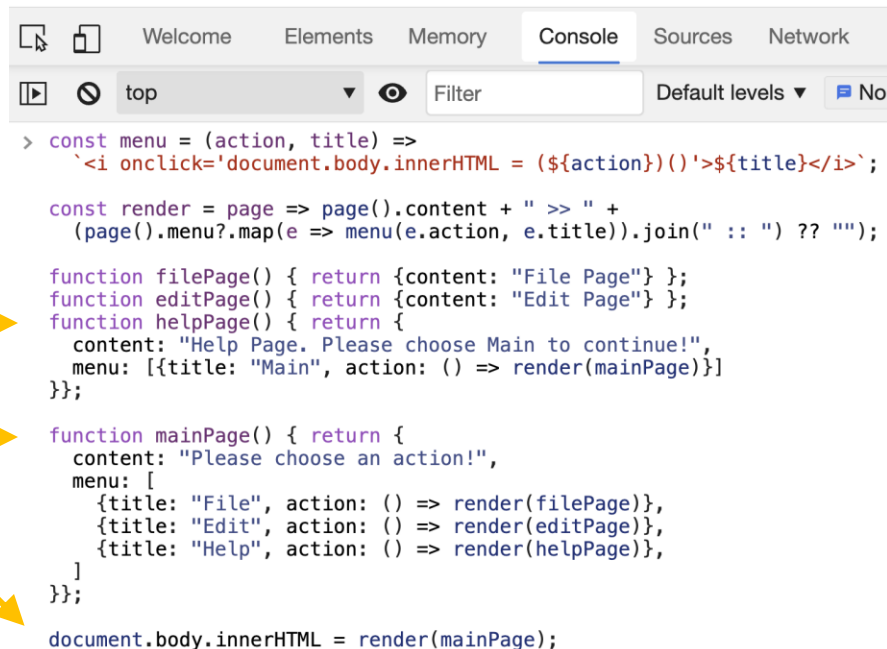
*helpPage depends
on mainPage*

*mainPage depends
on helpPage*

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

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*



```
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(" :: ") ?? "");

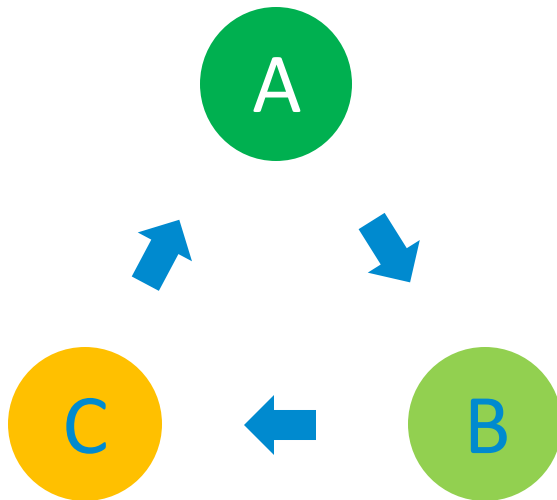
function filePage() { return {content: "File Page"} };
function editPage() { return {content: "Edit Page"} };
function helpPage() { return {
  content: "Help Page. Please choose Main to continue!",
  menu: [{title: "Main", action: () => render(mainPage)}]
}};

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);
```

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.

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

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

```
> 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 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

- it is [easier handle smaller files](#) (overview, navigate, edit)
- the tree structure of the directory system provides a visually clear [hierarchy between modules](#)
- [concerns are separated](#), harder do write spaghetti code
- [interfaces and dependencies are clearly visible](#)
- team members can [work independently](#) on different code parts

but breaking down into modules [without reasons could be problematic](#) as well

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

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](#).

```
1 <!DOCTYPE html>
2 <head>
3   <title>Il etait une fois un traditionnel</title>
4 </head>
5
6 <body>
7   <script src="src/common.js"></script>
8   <script src="src/file-page.js"></script>
9   <script src="src/edit-page.js"></script>
10  <script src="src/help-page.js"></script>
11  <script src="src/main-page.js"></script>
12 </body>

```

```
head
common.js x
1 const menu = (action, title) =>
2   `<i onclick='document.body.innerHTML = (${action})()'>${title}</i>`;
3
4 const render = page => page().content + " >> " +
5   (page().menu?.map(e => menu(e.action, e.title)).join(" :: ") ?? "");

```

```
file-page.js x edit-page.js x help-page.js x
1 function helpPage() { return {
2   content: "Help Page. Please choose Main to continue!",
3   menu: [{title: "Main", action: () => render(mainPage)}]
4 };

```

```
main-page.js x
1 function mainPage() { return {
2   content: "Please choose an action!",
3   menu: [
4     {title: "File", action: () => render(filePage)},
5     {title: "Edit", action: () => render(editPage)},
6     {title: "Help", action: () => render(helpPage)},
7   ]
8 };
9
10 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](#)

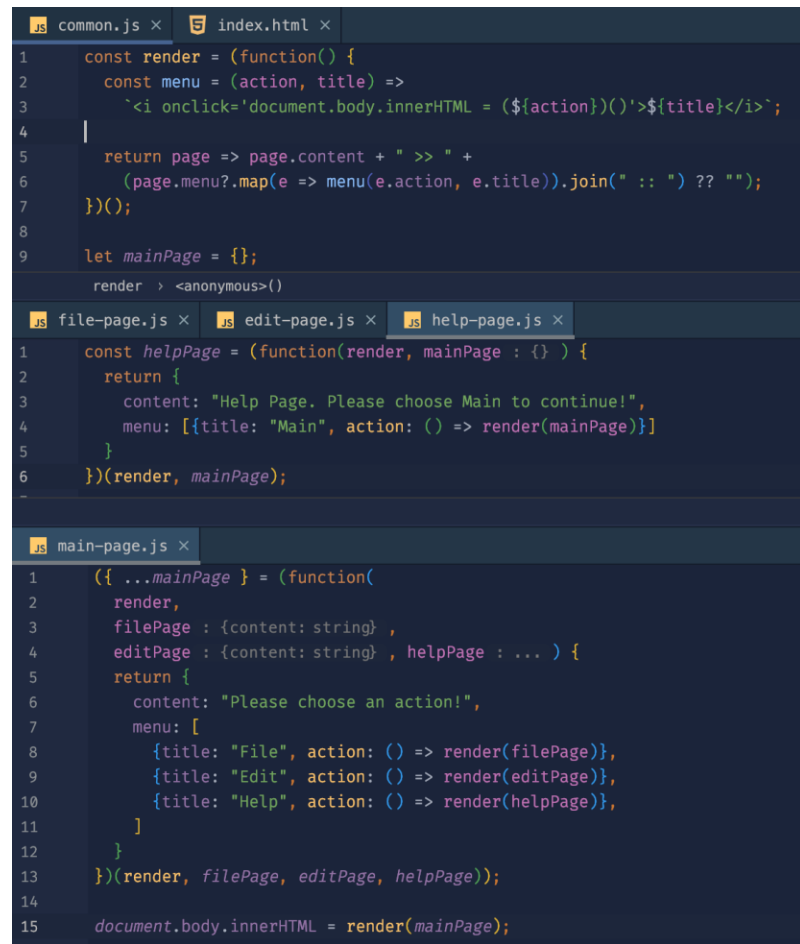
This simple solution utilizes the IIFE

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.



```
common.js
1  const render = (function() {
2    const menu = (action, title) =>
3      `<i onclick='document.body.innerHTML = (${action})()'>${title}</i>`;
4
5    return page => page.content + " >> " +
6      (page.menu?.map(e => menu(e.action, e.title)).join(" :: ") ?? "");
7  })();
8
9  let mainPage = {};

render > <anonymous>()

file-page.js
1  const helpPage = (function(render, mainPage : {} ) {
2    return {
3      content: "Help Page. Please choose Main to continue!",
4      menu: [{title: "Main", action: () => render(mainPage)}]
5    }
6  })(render, mainPage);

main-page.js
1  ({ ...mainPage } = (function(
2    render,
3    filePage : {content: string} ,
4    editPage : {content: string} , helpPage : ... ) {
5    return {
6      content: "Please choose an action!",
7      menu: [
8        {title: "File", action: () => render(filePage)},
9        {title: "Edit", action: () => render(editPage)},
10       {title: "Help", action: () => render(helpPage)},
11     ]
12   }
13 })(render, filePage, editPage, helpPage));
14
15 document.body.innerHTML = render(mainPage);
```

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

The screenshot shows a code editor with a project structure on the left and three open JavaScript files. The project structure includes a 'src' directory with a 'pages' subdirectory containing 'common.js', 'edit-page.js', 'file-page.js', 'help-page.js', and 'main-page.js'. The 'common.js' file defines a 'render' function. The 'edit-page.js' file defines a 'content' string and an 'action' array. The 'main-page.js' file uses 'require' to import the other modules and defines a 'mainPage' object with 'content' and 'actions'.

```
1 module.exports = {
2   render: function(page) {
3     return page.content;
4   }
5 };

1 module.exports = {
2   content: "Edit page",
3   action: []
4 };

1 const render = require("./common").render;
2 const filePage = require("./file-page");
3 const editPage = require("./edit-page");
4 const helpPage = require("./help-page");
5
6 const mainPage = {
7   content: "Press any key to go to the main page!",
8   actions: [
9     render(filePage),
10    render(editPage),
11    render(helpPage),
12  ]
13 };
14
15 module.exports = mainPage;
16
17 console.log(render(mainPage));
```

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



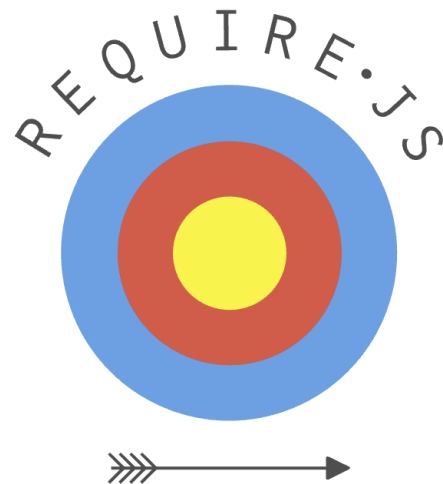
```
~/il-etait-une-fois> node src/pages/main-page.js  
Press any key to go to the main page!  
(node:97071) Warning: Accessing non-existent property 'content' of module exports inside circular dependency  
(Use `node --trace-warnings ...` to show where the warning was created)
```

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 a good
idea to utilize global variables in a module

The screenshot displays a code editor with several files open, illustrating ES6 module syntax. The `index.html` file (lines 1-14) includes a DOCTYPE declaration, a head section with a title and several `<script>` tags, and a body section. The script tags use `type="module"` and `src` attributes to load `pages/common.js`, `pages/file-page.js`, `pages/edit-page.js`, `pages/help-page.js`, and `index.js`. The `index.js` file (lines 1-4) imports `render` from `./pages/common.js` and `mainPage` from `./pages/main-page.js`, then sets `document.body.innerHTML = render(mainPage);`. The `common.js` file (lines 1-7) defines a `menu` function, a `render` function, and assigns `globalThis.render = render;`. The `file-page.js` file (lines 1-11) imports `render` and `mainPage`, defines a `helpPage` object, and exports it with `export default helpPage;` and `globalThis.helpPage = helpPage;`. The `edit-page.js` and `help-page.js` files are also open but their content is not fully visible. The `main-page.js` file (lines 1-16) imports `render` and the three page objects, defines a `mainPage` object, and exports it with `export default mainPage;` and `globalThis.mainPage = mainPage;`. Yellow arrows point from the text annotations on the left to specific code elements: from the first paragraph to the `type="module"` attribute in `index.html`; from the second paragraph to the `nomodule` attribute (though not present in the image, the arrow points to the script tag area); from the third paragraph to the `import` statements in `index.js` and `file-page.js`; from the fourth paragraph to the `export default` and `globalThis` usage in `file-page.js` and `main-page.js`.

```
index.html x
1 <!DOCTYPE html>
2 <head>
3   <title>Il était une fois un Modules ES6</title>
4   <script type="module" src="pages/common.js"></script>
5   <script type="module" src="pages/file-page.js"></script>
6   <script type="module" src="pages/edit-page.js"></script>
7   <script type="module" src="pages/help-page.js"></script>
8   <script type="module" src="pages/main-page.js"></script>
9   <script type="module" src="index.js"></script>
10 </head>
11 <body>
12 </body>
13 </body>
14

index.js x
1 import { render } from "./pages/common.js";
2 import mainPage from "./pages/main-page.js";
3
4 document.body.innerHTML = render(mainPage);

common.js x
1 const menu = (action, title) =>
2   `<i onclick="document.body.innerHTML = (${action})()>${title}</i>`;
3
4 export const render = page => page.content + " >> " +
5   (page.menu?.map(e => menu(e.action, e.title)).join(" :: ") ?? "");
6
7 globalThis.render = render;

file-page.js x edit-page.js x help-page.js x main-page.js x
1 import { render } from "./common.js";
2 import mainPage from "./main-page.js";
3
4 const helpPage = {
5   content: "Help Page. Please choose Main to continue!",
6   menu: [{title: "Main", action: () => render(mainPage)}]
7 };
8
9 export default helpPage;
10 globalThis.helpPage = helpPage;
11

1 import { render } from "./common.js";
2 import filePage from "./file-page.js";
3 import editPage from "./edit-page.js";
4 import helpPage from "./help-page.js";
5
6 const mainPage = {
7   content: "Please choose an action!",
8   menu: [
9     {title: "File", action: () => render(filePage)},
10    {title: "Edit", action: () => render(editPage)},
11    {title: "Help", action: () => render(helpPage)},
12  ]
13 };
14
15 export default mainPage;
16 globalThis.mainPage = mainPage;
```

```
index.html x
1 <!DOCTYPE html>
2 <head>
3   <title>Il était une fois un Modules ES6</title>
4   <script type="module" src="pages/common.js"></script>
5   <script type="module" src="pages/file-page.js"></script>
6   <script type="module" src="pages/edit-page.js"></script>
7   <script type="module" src="pages/help-page.js"></script>
8   <script type="module" src="pages/main-page.js"></script>
9   <script type="module" src="index.js"></script>
10 </head>
11
12 <body>
13 </body>
14
index.js x
1 import { render } from "./pages/common.js";
2 import mainPage from "./pages/main-page.js";
3
4 document.body.innerHTML = render(mainPage);

common.js x
1 const menu = (action, title) =>
2   `<i onclick='document.body.innerHTML = (${action})()'>${title}</i>`;
3
4 export const render = page => page.content + " >> " +
5   (page.menu?.map(e => menu(e.action, e.title)).join(" :: ") ?? "");
6
7 globalThis.render = render;

file-page.js x edit-page.js x help-page.js x main-page.js x
1 import { render } from "../common.js";
2 import mainPage from "../main-page.js";
3
4 const helpPage = {
5   content: "Help Page. Please choose Main to continue!",
6   menu: [{title: "Main", action: () => render(mainPage)}]
7 };
8
9 export default helpPage;
10 globalThis.helpPage = helpPage;
11
1 import { render } from "../common.js";
2 import filePage from "../file-page.js";
3 import editPage from "../edit-page.js";
4 import helpPage from "../help-page.js";
5
6 const mainPage = {
7   content: "Please choose an action!",
8   menu: [
9     {title: "File", action: () => render(filePage)},
10    {title: "Edit", action: () => render(editPage)},
11    {title: "Help", action: () => render(helpPage)},
12  ]
13 };
14
15 export default mainPage;
16 globalThis.mainPage = mainPage;
```

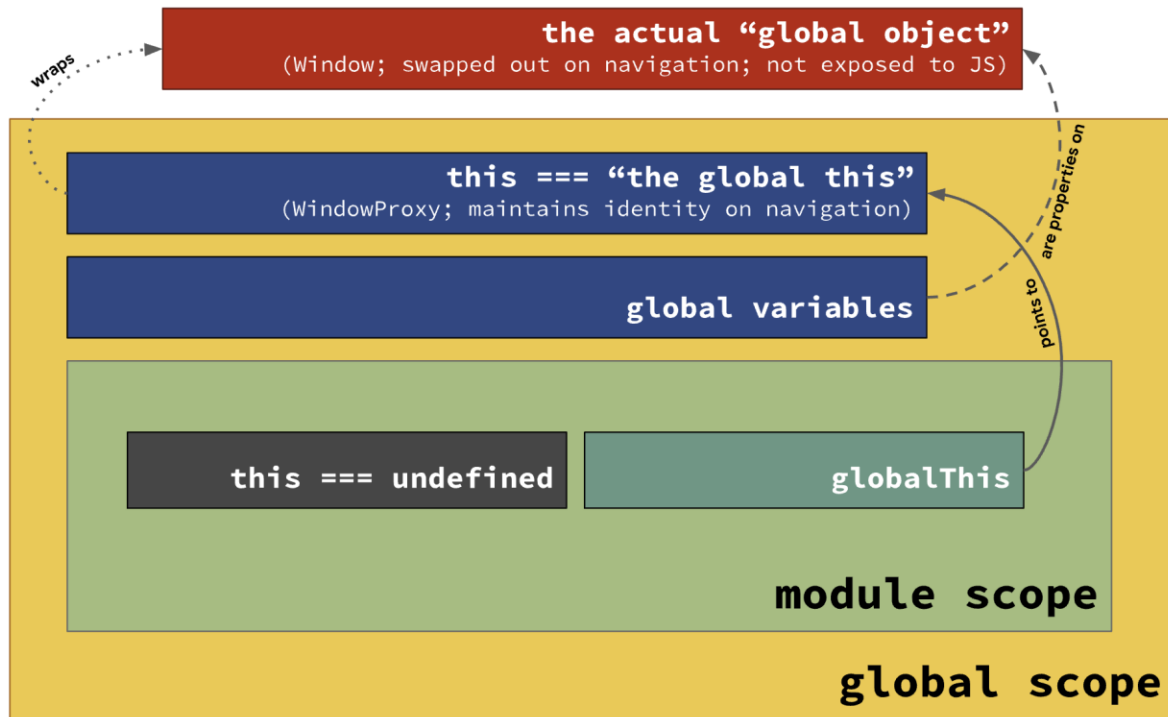
exports can be **named**...

importing **named** exports...
...and **default** exports

...or **default**

Q&A

Appendix



there is an excellent [article](#) on why `globalThis` is not a trivial thing