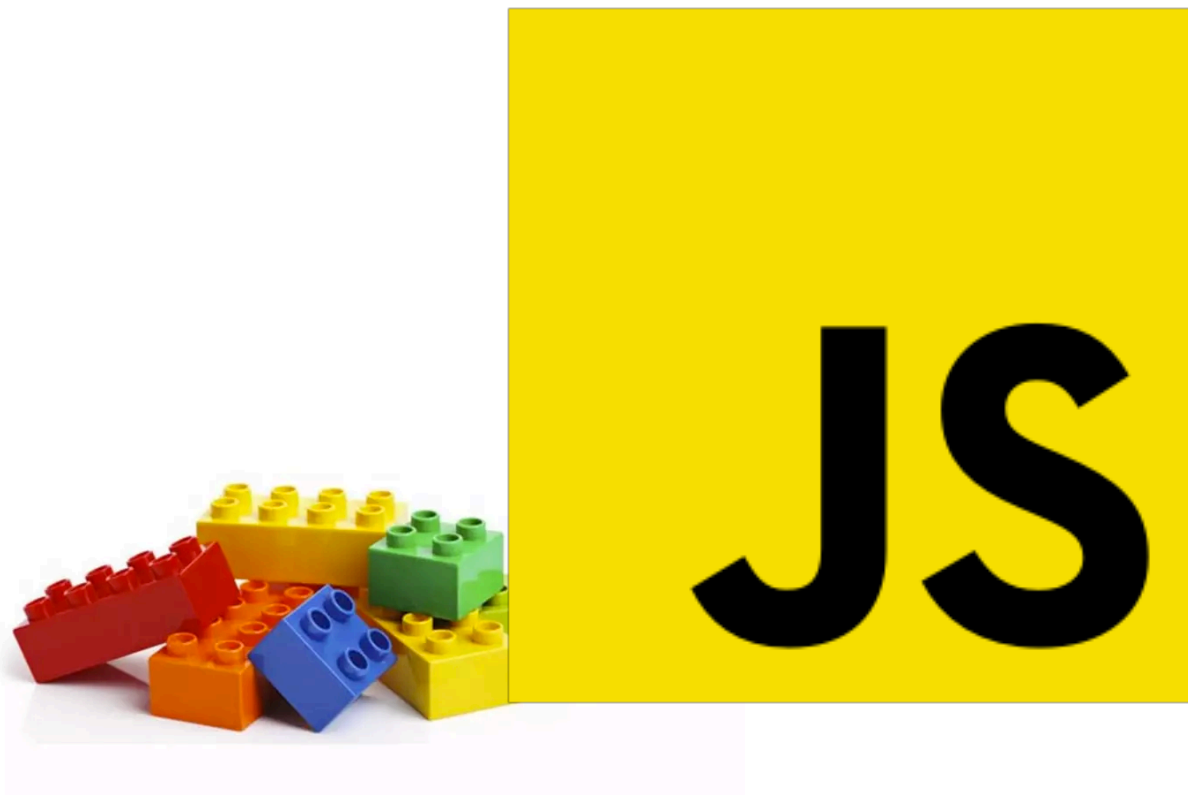


# Lesson - 28

Modules



# Lesson Plan

- HW Review
- Modules Introduction
- Import / Export
- Dynamic imports

# Modules Introduction

As our application grows bigger, we want to split it into multiple files, so called “modules”. A module usually contains a class or a library of functions.

For a long time, JavaScript existed without a language-level module syntax. That wasn't a problem, because initially scripts were small and simple, so there was no need.

But eventually scripts became more and more complex, so the community invented a variety of ways to organize code into modules, special libraries to load modules on demand.

For instance:

- [AMD](#) – one of the most ancient module systems, initially implemented by the library [require.js](#).
- [CommonJS](#) – the module system created for Node.js server.
- [UMD](#) – one more module system, suggested as a universal one, compatible with AMD and CommonJS.

Now all these slowly become a part of history, but we still can find them in old scripts.

The language-level module system appeared in the standard in 2015, gradually evolved since then, and is now supported by all major browsers and in Node.js. So we'll study it from now on.

# What is a module?

A module is just a file. One script is one module.

Modules can load each other and use special directives `export` and `import` to interchange functionality, call functions of one module from another one:

- `export` keyword labels variables and functions that should be accessible from outside the current module.
- `import` allows the import of functionality from other modules.

For instance, if we have a file `sayHi.js` exporting a function:

```
1 // 📁 sayHi.js
2 export function sayHi(user) {
3   alert(`Hello, ${user}!`);
4 }
```

...Then another file may import and use it:

```
1 // 📁 main.js
2 import {sayHi} from './sayHi.js';
3
4 alert(sayHi); // function...
5 sayHi('John'); // Hello, John!
```

The `import` directive loads the module by path `./sayHi.js` relative to the current file, and assigns exported function `sayHi` to the corresponding variable.

# Only via HTTP

 **Modules work only via HTTP, not in local files**

If you try to open a web-page locally, via `file://` protocol, you'll find that `import/export` directives don't work. Use a local web-server, such as [static-server](#) or use the "live server" capability of your editor, such as VS Code [Live Server Extension](#) to test them.

# Core modules features

What's different in modules, compared to "regular" scripts?

There are core features, valid both for browser and server-side JavaScript.

## Always "use strict"

Modules always `use strict`, by default. E.g. assigning to an undeclared variable will give an error.

```
1 <script type="module">
2   a = 5; // error
3 </script>
```



# Core modules features

## Module-level scope

Each module has its own top-level scope. In other words, top-level variables and functions from a module are not seen in other scripts.

In the example below, two scripts are imported, and `hello.js` tries to use `user` variable declared in `user.js`, and fails:



```
// file user.js
let user = "John";

// file hello.js
alert(user); // no such variable (each module has independent variables)

// index.html
<!doctype html>
<script type="module" src="user.js"></script>
<script type="module" src="hello.js"></script>
```

# Core modules features

Modules are expected to `export` what they want to be accessible from outside and `import` what they need.

So we should import `user.js` into `hello.js` and get the required functionality from it instead of relying on global variables.

This is the correct variant:



```
// file user.js
export let user = "John";

// file hello.js
import {user} from './user.js';

document.body.innerHTML = user; // John

// index.html
<!doctype html>
<script type="module" src="hello.js"></script>
```

If we really need to make a window-level global variable, we can explicitly assign it to `window` and access as `window.user`. But that's an exception requiring a good reason.



# Core modules features

## A module code is evaluated only the first time when imported

If the same module is imported into multiple other places, its code is executed only the first time, then exports are given to all importers.

That has important consequences. Let's look at them using examples:

First, if executing a module code brings side-effects, like showing a message, then importing it multiple times will trigger it only once – the first time:

```
1 // 📁 alert.js
2 alert("Module is evaluated!");
```

```
1 // Import the same module from different files
2
3 // 📁 1.js
4 import `./alert.js`; // Module is evaluated!
5
6 // 📁 2.js
7 import `./alert.js`; // (shows nothing)
```

# Core modules features

In practice, top-level module code is mostly used for initialization, creation of internal data structures, and if we want something to be reusable – export it.

Now, a more advanced example.

Let's say, a module exports an object:

```
1 // 📁 admin.js
2 export let admin = {
3   name: "John"
4 };
```

If this module is imported from multiple files, the module is only evaluated the first time, `admin` object is created, and then passed to all further importers.

All importers get exactly the one and only `admin` object:

```
1 // 📁 1.js
2 import {admin} from './admin.js';
3 admin.name = "Pete";
4
5 // 📁 2.js
6 import {admin} from './admin.js';
7 alert(admin.name); // Pete
8
9 // Both 1.js and 2.js imported the same object
10 // Changes made in 1.js are visible in 2.js
```

# Core modules features

For instance, the `admin.js` module may provide certain functionality, but expect the credentials to come into the `admin` object from outside:

```
1 // 📁 admin.js
2 export let admin = { };
3
4 export function sayHi() {
5   alert(`Ready to serve, ${admin.name}!`);
6 }
```

In `init.js`, the first script of our app, we set `admin.name`. Then everyone will see it, including calls made from inside `admin.js` itself:

```
1 // 📁 init.js
2 import {admin} from './admin.js';
3 admin.name = "Pete";
```

# Core modules features

## In a module, "this" is undefined

That's kind of a minor feature, but for completeness we should mention it.

In a module, top-level `this` is undefined.

Compare it to non-module scripts, where `this` is a global object:

```
1 <script>
2   alert(this); // window
3 </script>
4
5 <script type="module">
6   alert(this); // undefined
7 </script>
```

# Browser Specific features

There are also several browser-specific differences of scripts with `type="module"` compared to regular ones.

You may want skip this section for now if you're reading for the first time, or if you don't use JavaScript in a browser.

## Module scripts are deferred

Module scripts are *always* deferred, same effect as `defer` attribute (described in the chapter [Scripts: async, defer](#)), for both external and inline scripts.

In other words:

- downloading external module scripts `<script type="module" src="...">` doesn't block HTML processing, they load in parallel with other resources.
- module scripts wait until the HTML document is fully ready (even if they are tiny and load faster than HTML), and then run.
- relative order of scripts is maintained: scripts that go first in the document, execute first.

# Browser Specific features

As a side-effect, module scripts always “see” the fully loaded HTML-page, including HTML elements below them.

For instance:

```
1 <script type="module">
2   alert(typeof button); // object: the script can 'see' the button below
3   // as modules are deferred, the script runs after the whole page is loaded
4 </script>
5
6 Compare to regular script below:
7
8 <script>
9   alert(typeof button); // Error: button is undefined, the script can't see e
10  // regular scripts run immediately, before the rest of the page is processed
11 </script>
12
13 <button id="button">Button</button>
```

Please note: the second script actually runs before the first! So we'll see `undefined` first, and then `object`.

That's because modules are deferred, so we wait for the document to be processed. The regular script runs immediately, so we see its output first.

When using modules, we should be aware that the HTML page shows up as it loads, and JavaScript modules run after that, so the user may see the page before the JavaScript application is ready. Some functionality may not work yet. We should put “loading indicators”, or otherwise ensure that the visitor won't be confused by that.

# Browser Specific features

## Async works on inline scripts

For non-module scripts, the `async` attribute only works on external scripts. Async scripts run immediately when ready, independently of other scripts or the HTML document.

For module scripts, it works on inline scripts as well.

For example, the inline script below has `async`, so it doesn't wait for anything.

It performs the import (fetches `./analytics.js`) and runs when ready, even if the HTML document is not finished yet, or if other scripts are still pending.

That's good for functionality that doesn't depend on anything, like counters, ads, document-level event listeners.

```
1 <!-- all dependencies are fetched (analytics.js), and the script runs -->
2 <!-- doesn't wait for the document or other <script> tags -->
3 <script async type="module">
4   import {counter} from './analytics.js';
5
6   counter.count();
7 </script>
```



# Browser Specific features

## External scripts

External scripts that have `type="module"` are different in two aspects:

1. External scripts with the same `src` run only once:

```
1 <!-- the script my.js is fetched and executed only once -->
2 <script type="module" src="my.js"></script>
3 <script type="module" src="my.js"></script>
```

2. External scripts that are fetched from another origin (e.g. another site) require [CORS](#) headers, as described in the chapter [Fetch: Cross-Origin Requests](#). In other words, if a module script is fetched from another origin, the remote server must supply a header `Access-Control-Allow-Origin` allowing the fetch.

```
1 <!-- another-site.com must supply Access-Control-Allow-Origin -->
2 <!-- otherwise, the script won't execute -->
3 <script type="module" src="http://another-site.com/their.js"></script>
```

That ensures better security by default.



# Browser Specific features

## No “bare” modules allowed

In the browser, `import` must get either a relative or absolute URL. Modules without any path are called “bare” modules. Such modules are not allowed in `import`.

For instance, this `import` is invalid:

```
1 import {sayHi} from 'sayHi'; // Error, "bare" module
2 // the module must have a path, e.g. './sayHi.js' or wherever the module is
```

Certain environments, like Node.js or bundle tools allow bare modules, without any path, as they have their own ways for finding modules and hooks to fine-tune them. But browsers do not support bare modules yet.

## Compatibility, “nomodule”

Old browsers do not understand `type="module"`. Scripts of an unknown type are just ignored. For them, it's possible to provide a fallback using the `nomodule` attribute:

```
1 <script type="module">
2   alert("Runs in modern browsers");
3 </script>
4
5 <script nomodule>
6   alert("Modern browsers know both type=module and nomodule, so skip this")
7   alert("Old browsers ignore script with unknown type=module, but execute this")
8 </script>
```

# Build tools

In real-life, browser modules are rarely used in their “raw” form. Usually, we bundle them together with a special tool such as [Webpack](#) and deploy to the production server.

One of the benefits of using bundlers – they give more control over how modules are resolved, allowing bare modules and much more, like CSS/HTML modules.

Build tools do the following:

1. Take a “main” module, the one intended to be put in `<script type="module">` in HTML.
2. Analyze its dependencies: imports and then imports of imports etc.
3. Build a single file with all modules (or multiple files, that’s tunable), replacing native `import` calls with bundler functions, so that it works. “Special” module types like HTML/CSS modules are also supported.
4. In the process, other transformations and optimizations may be applied:
  - Unreachable code removed.
  - Unused exports removed (“tree-shaking”).
  - Development-specific statements like `console` and `debugger` removed.
  - Modern, bleeding-edge JavaScript syntax may be transformed to older one with similar functionality using [Babel](#).
  - The resulting file is minified (spaces removed, variables replaced with shorter names, etc).

If we use bundle tools, then as scripts are bundled together into a single file (or few files), `import/export` statements inside those scripts are replaced by special bundler functions. So the resulting “bundled” script does not contain any `import/export`, it doesn’t require `type="module"`, and we can put it into a regular script:

```
1 <!-- Assuming we got bundle.js from a tool like Webpack -->
2 <script src="bundle.js"></script>
```

# Summary

To summarize, the core concepts are:

1. A module is a file. To make `import/export` work, browsers need `<script type="module">`. Modules have several differences:
  - Deferred by default.
  - Async works on inline scripts.
  - To load external scripts from another origin (domain/protocol/port), CORS headers are needed.
  - Duplicate external scripts are ignored.
2. Modules have their own, local top-level scope and interchange functionality via `import/export`.
3. Modules always `use strict`.
4. Module code is executed only once. Exports are created once and shared between importers.

When we use modules, each module implements the functionality and exports it. Then we use `import` to directly import it where it's needed. The browser loads and evaluates the scripts automatically.

In production, people often use bundlers such as [Webpack](#) to bundle modules together for performance and other reasons.

In the next chapter we'll see more examples of modules, and how things can be exported/imported.

# Export and Import

Export and import directives have several syntax variants.

In the previous article we saw a simple use, now let's explore more examples.

## Export before declarations

We can label any declaration as exported by placing `export` before it, be it a variable, function or a class.

For instance, here all exports are valid:

```
1 // export an array
2 export let months = ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'I
3
4 // export a constant
5 export const MODULES_BECAME_STANDARD_YEAR = 2015;
6
7 // export a class
8 export class User {
9     constructor(name) {
10         this.name = name;
11     }
12 }
```

### **i** No semicolons after export class/function

Please note that `export` before a class or a function does not make it a **function expression**. It's still a function declaration, albeit exported.

Most JavaScript style guides don't recommend semicolons after function and class declarations.

That's why there's no need for a semicolon at the end of `export class` and `export function`:

```
1 export function sayHi(user) {  
2   alert(`Hello, ${user}!`);  
3 } // no ; at the end
```

# Export apart from declarations

Also, we can put `export` separately.

Here we first declare, and then export:

```
1 // 📁 say.js
2 function sayHi(user) {
3   alert(`Hello, ${user}!`);
4 }
5
6 function sayBye(user) {
7   alert(`Bye, ${user}!`);
8 }
9
10 export {sayHi, sayBye}; // a list of exported variables
```

...Or, technically we could put `export` above functions as well.

# Import

Usually, we put a list of what to import in curly braces `import {...}`, like this:

```
1 // 📁 main.js
2 import {sayHi, sayBye} from './say.js';
3
4 sayHi('John'); // Hello, John!
5 sayBye('John'); // Bye, John!
```

But if there's a lot to import, we can import everything as an object using `import * as <obj>`, for instance:

```
1 // 📁 main.js
2 import * as say from './say.js';
3
4 say.sayHi('John');
5 say.sayBye('John');
```

At first sight, “import everything” seems such a cool thing, short to write, why should we ever explicitly list what we need to import?



# Import

Well, there are few reasons.

1. Modern build tools ([webpack](#) and others) bundle modules together and optimize them to speedup loading and remove unused stuff.

Let's say, we added a 3rd-party library `say.js` to our project with many functions:

```
1 // 📁 say.js
2 export function sayHi() { ... }
3 export function sayBye() { ... }
4 export function becomeSilent() { ... }
```

Now if we only use one of `say.js` functions in our project:

```
1 // 📁 main.js
2 import {sayHi} from './say.js';
```

...Then the optimizer will see that and remove the other functions from the bundled code, thus making the build smaller. That is called "tree-shaking".

2. Explicitly listing what to import gives shorter names: `sayHi()` instead of `say.sayHi()`.
3. Explicit list of imports gives better overview of the code structure: what is used and where. It makes code support and refactoring easier.




# Import

## Import "as"

We can also use `as` to import under different names.

For instance, let's import `sayHi` into the local variable `hi` for brevity, and import `sayBye` as `bye`:

```
1 //  main.js
2 import {sayHi as hi, sayBye as bye} from './say.js';
3
4 hi('John'); // Hello, John!
5 bye('John'); // Bye, John!
```

# Export

## Export "as"

The similar syntax exists for `export`.

Let's export functions as `hi` and `bye`:

```
1 // 📁 say.js
2 ...
3 export {sayHi as hi, sayBye as bye};
```

Now `hi` and `bye` are official names for outsiders, to be used in imports:

```
1 // 📁 main.js
2 import * as say from './say.js';
3
4 say.hi('John'); // Hello, John!
5 say.bye('John'); // Bye, John!
```

# Export default

In practice, there are mainly two kinds of modules.

1. Modules that contain a library, pack of functions, like `say.js` above.
2. Modules that declare a single entity, e.g. a module `user.js` exports only `class User`.

Mostly, the second approach is preferred, so that every “thing” resides in its own module.

Naturally, that requires a lot of files, as everything wants its own module, but that’s not a problem at all. Actually, code navigation becomes easier if files are well-named and structured into folders.

Modules provide a special `export default` (“the default export”) syntax to make the “one thing per module” way look better.

Put `export default` before the entity to export:

```
1 // 📁 user.js
2 export default class User { // just add "default"
3   constructor(name) {
4     this.name = name;
5   }
6 }
```

There may be only one `export default` per file.

# Export default

...And then import it without curly braces:

```
1 // 📁 main.js
2 import User from './user.js'; // not {User}, just User
3
4 new User('John');
```

Imports without curly braces look nicer. A common mistake when starting to use modules is to forget curly braces at all. So, remember, `import` needs curly braces for named exports and doesn't need them for the default one.

## Named export

```
export class User {...}
```

```
import {User} from ...
```

## Default export

```
export default class User {...}
```

```
import User from ...
```

Technically, we may have both default and named exports in a single module, but in practice people usually don't mix them. A module has either named exports or the default one.

As there may be at most one default export per file, the exported entity may have no name.

# Export default

For instance, these are all perfectly valid default exports:

```
1 export default class { // no class name
2   constructor() { ... }
3 }
```

```
1 export default function(user) { // no function name
2   alert(`Hello, ${user}!`);
3 }
```

```
1 // export a single value, without making a variable
2 export default ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
```

Not giving a name is fine, because there is only one `export default` per file, so `import` without curly braces knows what to import.

Without `default`, such an export would give an error:

```
1 export class { // Error! (non-default export needs a name)
2   constructor() {}
3 }
```

# The default name

In some situations the `default` keyword is used to reference the default export.

For example, to export a function separately from its definition:

```
1 function sayHi(user) {  
2   alert(`Hello, ${user}!`);  
3 }  
4  
5 // same as if we added "export default" before the function  
6 export {sayHi as default};
```

Or, another situation, let's say a module `user.js` exports one main "default" thing, and a few named ones (rarely the case, but it happens):

```
1 // 📁 user.js  
2 export default class User {  
3   constructor(name) {  
4     this.name = name;  
5   }  
6 }  
7  
8 export function sayHi(user) {  
9   alert(`Hello, ${user}!`);  
10 }
```

# A word against default exports

Named exports are explicit. They exactly name what they import, so we have that information from them; that's a good thing.

Named exports force us to use exactly the right name to import:

```
1 import {User} from './user.js';  
2 // import {MyUser} won't work, the name must be {User}
```

...While for a default export, we always choose the name when importing:

```
1 import User from './user.js'; // works  
2 import MyUser from './user.js'; // works too  
3 // could be import Anything... and it'll still work
```

So team members may use different names to import the same thing, and that's not good.

Usually, to avoid that and keep the code consistent, there's a rule that imported variables should correspond to file names, e.g:

```
1 import User from './user.js';  
2 import LoginForm from './loginForm.js';  
3 import func from '/path/to/func.js';  
4 ...
```

# Summary

Here are all types of `export` that we covered in this and previous articles.

You can check yourself by reading them and recalling what they mean:

- Before declaration of a class/function/...:
  - `export [default] class/function/variable ...`
- Standalone export:
  - `export {x [as y], ...}.`
- Re-export:
  - `export {x [as y], ...} from "module"`
  - `export * from "module"` (doesn't re-export default).
  - `export {default [as y]} from "module"` (re-export default).

Import:

- Named exports from module:
  - `import {x [as y], ...} from "module"`
- Default export:
  - `import x from "module"`
  - `import {default as x} from "module"`
- Everything:
  - `import * as obj from "module"`
- Import the module (its code runs), but do not assign it to a variable:
  - `import "module"`

We can put `import/export` statements at the top or at the bottom of a script, that doesn't matter.



# Dynamic imports

Export and import statements that we covered in previous chapters are called “static”. The syntax is very simple and strict.

First, we can’t dynamically generate any parameters of `import`.

The module path must be a primitive string, can’t be a function call. This won’t work:

```
1 import ... from getName(); // Error, only from "string" is allowed
```

Second, we can’t import conditionally or at run-time:

```
1 if(...) {  
2   import ...; // Error, not allowed!  
3 }  
4  
5 {  
6   import ...; // Error, we can't put import in any block  
7 }
```

That’s because `import / export` aim to provide a backbone for the code structure. That’s a good thing, as code structure can be analyzed, modules can be gathered and bundled into one file by special tools, unused exports can be removed (“tree-shaken”). That’s possible only because the structure of imports/exports is simple and fixed.

But how can we import a module dynamically, on-demand?

# The `import()` expression

The `import(module)` expression loads the module and returns a promise that resolves into a module object that contains all its exports. It can be called from any place in the code.

We can use it dynamically in any place of the code, for instance:

```
1 let modulePath = prompt("Which module to load?");
2
3 import(modulePath)
4   .then(obj => <module object>)
5   .catch(err => <loading error, e.g. if no such module>)
```

Or, we could use `let module = await import(modulePath)` if inside an async function.

For instance, if we have the following module `say.js`:

```
1 // 📁 say.js
2 export function hi() {
3   alert(`Hello`);
4 }
5
6 export function bye() {
7   alert(`Bye`);
8 }
```

**i Please note:**

Dynamic imports work in regular scripts, they don't require `script type="module"`.

**i Please note:**

Although `import()` looks like a function call, it's a special syntax that just happens to use parentheses (similar to `super()`).

So we can't copy `import` to a variable or use `call/apply` with it. It's not a function.

# Home work

Using import/export improve APOD (A picture of the day) project:

- Using import/export syntax refactor code, by extracting Apod's constructors methods to utils.js file(ex: **addElement**, **renderPage**, **dateFormatter**).
- Extract Apod constructor function, into separate file: apod.js
- Create index.js and import Apod from apod.js, then initialize it, and make new object available globally(window object).

# Learning Resources

## 1. Modules:

1. <https://javascript.info/modules-intro>
2. <https://javascript.info/import-export>
3. <https://javascript.info/modules-dynamic-imports>