8th February 2020

Export and Import

Export and import directives have several syntax variants.

In the previous chapter 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:

```
// export an array
export let months = ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', '

// export a constant
export const MODULES_BECAME_STANDARD_YEAR = 2015;

// export a class
export class User {
  constructor(name) {
    this.name = name;
  }
}
```

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`:

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

Export apart from declarations

Also, we can put `export` separately.

Here we first declare, and then export:

```
// 📁 say.js
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

function sayBye(user) {
  alert(`Bye, ${user}!`);
}

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:

```
// 📁 main.js
import {sayHi, sayBye} from './say.js';

sayHi('John'); // Hello, John!
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:

```
// 📁 main.js
import * as say from './say.js';

say.sayHi('John');
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?

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:

   ```
   // 📁 say.js
   export function sayHi() { ... }
   export function sayBye() { ... }
   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 “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 “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 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:

```
// 📁 user.js
export default class User { // just add "default"
  constructor(name) {
    this.name = name;
  }
}
```

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

…And then import it without curly braces:

```
// 📁 main.js
import User from './user.js'; // not {User}, just User

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	Default export
`export class User {...}`	`export default class User {...}`
`import {User} from ...`	`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.

For instance, these are all perfectly valid default exports:

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

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

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

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

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

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

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:

```
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

// same as if we added "export default" before the function
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 happens):

```
// 🗂 user.js
export default class User {
  constructor(name) {
    this.name = name;
  }
}

export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

Here’s how to import the default export along with a named one:

```
// 🗂 main.js
import {default as User, sayHi} from './user.js';

new User('John');
```

And, finally, if importing everything `*` as an object, then the `default` property is exactly the default export:

```
// 📁 main.js
import * as user from './user.js';

let User = user.default; // the default export
new User('John');
```

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 enforce us to use exactly the right name to import:

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

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

```
import User from './user.js'; // works
import MyUser from './user.js'; // works too
// 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:

```
import User from './user.js';
import LoginForm from './loginForm.js';
import func from '/path/to/func.js';
...
```

Still, some teams consider it a serious drawback of default exports. So they prefer to always use named exports. Even if only a single thing is exported, it's still exported under a name, without `default`.

That also makes re-export (see below) a little bit easier.

Re-export

"Re-export" syntax `export ... from ...` allows to import things and immediately export them (possibly under another name), like this:

```
export {sayHi} from './say.js'; // re-export sayHi

export {default as User} from './user.js'; // re-export default
```

Why would that be needed? Let's see a practical use case.

Imagine, we're writing a "package": a folder with a lot of modules, with some of the functionality exported outside (tools like NPM allow to publish and distribute such packages), and many modules are just "helpers", for the internal use in other package modules.

The file structure could be like this:

```
auth/
    index.js
    user.js
    helpers.js
    tests/
        login.js
    providers/
        github.js
        facebook.js
        ...
```

We'd like to expose the package functionality via a single entry point, the "main file" `auth/index.js`, to be used like this:

```
import {login, logout} from 'auth/index.js'
```

The idea is that outsiders, developers who use our package, should not meddle with its internal structure, search for files inside our package folder. We export only what's necessary in `auth/index.js` and keep the rest hidden from prying eyes.

As the actual exported functionality is scattered among the package, we can import it into `auth/index.js` and export from it:

```
// 📁 auth/index.js

// import login/logout and immediately export them
import {login, logout} from './helpers.js';
export {login, logout};

// import default as User and export it
import User from './user.js';
export {User};
...
```

Now users of our package can `import {login} from "auth/index.js"`.

The syntax `export ... from ...` is just a shorter notation for such import-export:

```
// 📁 auth/index.js
// import login/logout and immediately export them
export {login, logout} from './helpers.js';

// import default as User and export it
export {default as User} from './user.js';
...
```

Re-exporting the default export

The default export needs separate handling when re-exporting.

Let's say we have `user.js`, and we'd like to re-export class `User` from it:

```
// 🗂 user.js
export default class User {
  // ...
}
```

1. `export User from './user.js'` won't work. What can go wrong?... But that's a syntax error!

   To re-export the default export, we have to write `export {default as User}`, as in the example above.

2. `export * from './user.js'` re-exports only named exports, but ignores the default one.

   If we'd like to re-export both named and the default export, then two statements are needed:

```
export * from './user.js'; // to re-export named exports
export {default} from './user.js'; // to re-export the default export
```

Such oddities of re-exporting the default export are one of the reasons why some developers don't like them.

Summary

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

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.

So, technically this code is fine:

```
sayHi();

// ...

import {sayHi} from './say.js'; // import at the end of the file
```

In practice imports are usually at the start of the file, but that's only for more convenience.

Please note that import/export statements don't work if inside `{...}`.

A conditional import, like this, won't work:

```
if (something) {
  import {sayHi} from "./say.js"; // Error: import must be at top level
}
```

…But what if we really need to import something conditionally? Or at the right time? Like, load a module upon request, when it's really needed?

We'll see dynamic imports in the next chapter.

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)