Export and Import

Export before declarations

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

EXAMPLE

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

There are no semicolons after **export class/function** because most JavaScript style guides don’t recommend semicolons after function and class declarations and export before a class or a function does not make it a function expression. It’s still a function declaration.

Export Apart from declarations

It is also possible to put export separately. We first declare, and then export:

EXAMPLE

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

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

Import*

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

EXAMPLE

```
import {sayHi, sayBye} from './say.js';
sayHi('John') ;
sayBye('John') ;
```

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

```
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.

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

Don’t be afraid to import too much
Modern build tools, such as webpack and others, bundle modules together and optimize them to speedup loading. They also removed unused imports.
For instance, if you **`import * as`** `library` from a huge code library, and then use only few methods, then unused ones will not be included into the optimized bundle.

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 :

EXAMPLE

```
import {sayHi as hi, sayBye as bye} from './say.js';
 hi('John') ;
 bye('John') ;
```

Export as

EXAMPLE

The similar syntax exists for export. Let’s export functions as **hi** and **bye** : export {sayHi as **hi**, sayBye as **bye**}; Now **hi** and **bye** are official names for outsiders, to be used in imports :

```
import * as say from './say.js';
say.hi('John');
say.bye('John') ;
```

Export default

In practice, there are mainly two kinds of modules:

- Modules that contain a library, pack of functions, like say.js above.
- 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.

EXAMPLE

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:

EXAMPLE

```
// 📁 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 :

EXAMPLE

```
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 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:

```
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 it happens):

EXAMPLE

```
// 📁 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:

EXAMPLE

```
// 📁 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:

EXAMPLE

```
// 📁 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 force 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 us to publish and distribute such packages, but we don't have to use them), and many modules are just "helpers", for 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.
In other words, a person who would like to use our package, should import only from the "main file" auth/index.js.
Like this:
```
import {login, logout} from 'auth/index.js'
```
The "main file", auth/index.js exports all the functionality that we'd like to provide in our package.

The idea is that outsiders, other programmers 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:

EXAMPLE

```
// 📁 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:

EXAMPLE

```
// 📁 auth/index.js
// re-export login/logout
export {login, logout} from './helpers.js';

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

The notable difference of export ... from compared to import/export is that re-exported modules aren't available in the current file. So inside the above example of auth/index.js we can't use re-exported login/logout functions.

Re-exporting the default export

The default export needs separate handling when re-exporting.
Let's say we have user.js with the export default class User and would like to re-export it:

EXAMPLE

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

We can come across two problems with it:
export User from './user.js' won't work. That would lead to a syntax error.
To re-export the default export, we have to write export {default as User}, as in the example above.
export * from './user.js' re-exports only named exports, but ignores the default one.

If we'd like to re-export both named and default exports, 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 a default export are one of the reasons why some developers don't like default exports and prefer named ones.

Object methods, "this"

Actions are represented in JavaScript by functions in properties.
It's common that an object method needs to access the information stored in the object to do its job.
For instance, the code inside user.sayHi() may need the name of the user.
To access the object, a method can use the this keyword.
The value of this is the object "before dot", the one used to call the method.
For instance:

EXAMPLE

```
let user = {
  name: "John",
  age: 30,

  sayHi() {
    // "this" is the "current object"
    alert(this.name);
  }

};
user.sayHi(); // John
```

'this' is not bound

In JavaScript, keyword this behaves unlike most other programming languages. It can be used in any function, even if it's not a method of an object.
There's no syntax error in the following example:

EXAMPLE

```
function sayHi() {
  alert( this.name );
}
```

The consequences of unbound `this`

If you come from another programming language, then you are probably used to the idea of a "bound `this`", where methods defined in an object always have `this` referencing that object.

In JavaScript `this` is “free”, its value is evaluated at call-time and does not depend on where the method was declared, but rather on what object is “before the dot”.
The concept of run-time evaluated `this` has both pluses and minuses. On the one hand, a function can be reused for different objects. On the other hand, the greater flexibility creates more possibilities for mistakes.

Arrow functions have no this

Arrow functions are special: they don’t have their “own” this. If we reference this from such a function, it’s taken from the outer “normal” function.
For instance, here arrow() uses this from the outer user.sayHi() method:

```
let user = {
  firstName: "Ilya",
  sayHi() {
    let arrow = () => alert(this.firstName);
    arrow();
  }
};
user.sayHi(); // Ilya
```

That’s a special feature of arrow functions, it’s useful when we actually do not want to have a separate this, but rather to take it from the outer context.

Class basic syntax

The basic syntax is:

```
class MyClass {
// class methods
constructor() { ... }
method1() { ... }
method2() { ... }
method3() { ... }
...
}
```

Then use new MyClass() to create a new object with all the listed methods.
The constructor() method is called automatically by new, so we can initialize the object there.

For example:

```
class User {

constructor(name) {
this.name = name;
}
sayHi() {
alert(this.name);
}

}

// Usage:
let user = new User("John");
user.sayHi();
```

When new User("John") is called:
A new object is created.
The constructor runs with the given argument and assigns it to this.name.
…Then we can call object methods, such as user.sayHi().

No comma between class methods
A common pitfall for novice developers is to put a comma between class methods, which would result in a syntax error.
The notation here is not to be confused with object literals. Within the class, no commas are required.

What is a class

So, what exactly is a class? That's not an entirely new language-level entity, as one might think.
Let's unveil any magic and see what a class really is. That'll help in understanding many complex aspects.
In JavaScript, a class is a kind of function.
Here, take a look:

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

// proof: User is a function
alert(typeof User); // function
```

Making bound methods with class fields

It depends on the context of the call.
So if an object method is passed around and called in another context, this won't be a reference to its object any more.

For instance, this code will show undefined:

```
class Button {
  constructor(value) {
    this.value = value;
  }

  click() {
    alert(this.value);
  }
}

let button = new Button("hello");

setTimeout(button.click, 1000); // undefined
```

The problem is called "losing this".
There are two approache :

- Pass a wrapper-function, such as setTimeout(() => button.click(), 1000).
- Bind the method to object, e.g. in the constructor.

Class fields provide another, quite elegant syntax:

```
class Button {
  constructor(value) {
    this.value = value;
  }
  click = () => {
    alert(this.value);
  }
}

let button = new Button("hello");

setTimeout(button.click, 1000); // hello
```

The class field click = () => {...} is created on a per-object basis, there's a separate function for each Button object, with this inside it referencing that object. We can pass button.click around anywhere, and the value of this will always be correct.

LEARN MORE

Reading

1. Read 9.1 Class basic syntax.
2. Watch 4.4 Object Methods, "This". If the concept of `this` still is a bit unclear, you can continue with either this article: This in JavaScript OR this video: What is THIS keyword in JS (The concepts in the video are very well explained...but there are a couple of 'bleeped' out profanities!)
3. Read 13.2 Modules: Import and Export.