Экспорт и импорт

Директивы экспорт и импорт имеют несколько вариантов вызова.

В предыдущей главе мы видели простое использование, давайте теперь посмотрим больше примеров.

Экспорт до объявления

Мы можем пометить любое объявление как экспортируемое, разместив `export` перед ним, будь то переменная, функция или класс.

Например, все следующие экспорты допустимы:

```
// экспорт массива
export let months = ['Jan', 'Feb', 'Mar', 'Apr', 'Aug',

// экспорт константы
export const MODULES_BECAME_STANDARD_YEAR = 2015;

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

Не ставится точка с запятой после экспорта класса/функции

Обратите внимание, что `export` перед классом или функцией не делает их функциональным выражением. Это всё также объявление функции, хотя и экспортируемое.

Большинство руководств по стилю кода в JavaScript не рекомендуют ставить точку с запятой после объявлений функций или классов.

Поэтому в конце `export class` и `export function` не нужна точка с запятой:

```
export function sayHi(user) {
  alert(`Hello, ${user}!`);
}  // без ; в конце
```

Экспорт отдельно от объявления

Также можно написать `export` отдельно.

Здесь мы сначала объявляем, а затем экспортируем:

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

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

export {sayHi, sayBye}; // список экспортируемых переме
```

...Или, технически, мы также можем расположить `export` выше функций.

Импорт *

Обычно мы располагаем список того, что хотим импортировать, в фигурных скобках `import {...}`, например вот так:

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

sayHi('John'); // Hello, John!
sayBye('John'); // Bye, John!
```

Но если импортировать нужно много чего, мы можем импортировать всё сразу в виде объекта, используя `import * as <obj>`. Например:

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

say.sayHi('John');
say.sayBye('John');
```

На первый взгляд «импортировать всё» выглядит очень удобно, не надо писать лишнего, зачем нам вообще может понадобиться явно перечислять список того, что нужно импортировать?

Для этого есть несколько причин.

1. Современные инструменты сборки (webpack и другие) собирают модули вместе и оптимизируют их, ускоряя загрузку и удаляя неиспользуемый код.

   Предположим, мы добавили в наш проект стороннюю библиотеку `say.js` с множеством функций:

   ```
   // 🐬 say.js
   export function sayHi() { ... }
   export function sayBye() { ... }
   export function becomeSilent() { ... }
   ```

   Теперь, если из этой библиотеки в проекте мы используем только одну функцию:

   ```
   // 🐬 main.js
   import {sayHi} from './say.js';
   ```

   ...Тогда оптимизатор увидит, что другие функции не используются, и удалит остальные из собранного кода, тем самым делая код меньше. Это называется «tree-shaking».

2. Явно перечисляя то, что хотим импортировать, мы получаем более короткие имена функций: `sayHi()` вместо `say.sayHi()`.
3. Явное перечисление импортов делает код более понятным, позволяет увидеть, что именно и где используется. Это упрощает поддержку и рефакторинг кода.

Импорт «как»

Мы также можем использовать `as`, чтобы импортировать под другими именами.

Например, для краткости импортируем `sayHi` в локальную переменную `hi`, а `sayBye` импортируем как `bye`:

```
// 🐬 main.js
```


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

Экспортировать «как»

Аналогичный синтаксис существует и для `export`.

Давайте экспортируем функции, как `hi` и `bye`:

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

Теперь `hi` и `bye` – официальные имена для внешнего кода, их нужно использовать при импорте:

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

Экспорт по умолчанию

На практике модули встречаются в основном одного из двух типов:

1. Модуль, содержащий библиотеку или набор функций, как `say.js` выше.
2. Модуль, который объявляет что-то одно, например модуль `user.js` экспортирует только `class User`.

По большей части, удобнее второй подход, когда каждая «вещь» находится в своём собственном модуле.

Естественно, требуется много файлов, если для всего делать отдельный модуль, но это не проблема. Так даже удобнее: навигация по проекту становится проще, особенно, если у файлов хорошие имена, и они структурированы по папкам.

Модули предоставляют специальный синтаксис `export default` («экспорт по умолчанию») для второго подхода.

Ставим `export default` перед тем, что нужно экспортировать:

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

Заметим, в файле может быть не более одного `export default`.

...И потом импортируем без фигурных скобок:

```
1  // 🐚 main.js
2  import User from './user.js'; // не {User}, просто User
3
4  new User('John');
```

Импорты без фигурных скобок выглядят красивее. Обычная ошибка начинающих: забывать про фигурные скобки. Запомним: фигурные скобки необходимы в случае именованных экспортов, для `export default` они не нужны.

Именованный экспорт	Экспорт по умолчанию
`export class User {...}`	`export default class User {...}`
`import {User} from ...`	`import User from ...`

Технически в одном модуле может быть как экспорт по умолчанию, так и именованные экспорты, но на практике обычно их не смешивают. То есть, в модуле находятся либо именованные экспорты, либо один экспорт по умолчанию.

Так как в файле может быть максимум один `export default`, то экспортируемая сущность не обязана иметь имя.

Например, всё это – полностью корректные экспорты по умолчанию:

```
export default class { // у класса нет имени
  constructor() { ... }
}
```

```
export default function(user) { // у функции нет имени
  alert(`Hello, ${user}!`);
}
```

```
// экспортируем значение, не создавая переменную
export default ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep'
```

Это нормально, потому что может быть только один `export default` на файл, так что `import` без фигурных скобок всегда знает, что импортировать.

Без `default` такой экспорт выдал бы ошибку:

```
export class { // Ошибка! (необходимо имя, если это не
  constructor() {}
}
```

Имя «default»

В некоторых ситуациях для обозначения экспорта по умолчанию в качестве имени используется `default`.

Например, чтобы экспортировать функцию отдельно от её объявления:

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

// то же самое, как если бы мы добавили "export default
export {sayHi as default};
```

Или, ещё ситуация, давайте представим следующее: модуль `user.js` экспортирует одну сущность «по умолчанию» и несколько именованных (редкий, но возможный случай):

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

export function sayHi(user) {

```

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


Вот как импортировать экспорт по умолчанию вместе с именованным экспортом:

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

new User('John');
```

И, наконец, если мы импортируем всё как объект `import *`, тогда его свойство `default` – как раз и будет экспортом по умолчанию:

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

let User = user.default; // экспорт по умолчанию
new User('John');
```

Довод против экспортов по умолчанию

Именованные экспорты «включают в себя» своё имя. Эта информация является частью модуля, говорит нам, что именно экспортируется.

Именованные экспорты вынуждают нас использовать правильное имя при импорте:

```
import {User} from './user.js';
// import {MyUser} не сработает, должно быть именно имя
```

...В то время как для экспорта по умолчанию мы выбираем любое имя при импорте:

```
import User from './user.js'; // сработает
import MyUser from './user.js'; // тоже сработает
// можно импортировать с любым именем, и это будет рабо
```

Так что члены команды могут использовать разные имена для импорта одной и той же вещи, и это не очень хорошо.

Обычно, чтобы избежать этого и соблюсти единообразие кода, есть правило: имена импортируемых переменных должны соответствовать именам файлов. Вот так:

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

Тем не менее, в некоторых командах это считают серьёзным доводом против экспортов по умолчанию и предпочитают использовать именованные экспорты везде. Даже если экспортируется только одна вещь, она всё равно экспортируется с именем, без использования `default`.

Это также немного упрощает реэкспорт (смотрите ниже).

Реэкспорт

Синтаксис «реэкспорта» `export ... from ...` позволяет импортировать что-то и тут же экспортировать, возможно под другим именем, вот так:

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

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

Зачем это нужно? Рассмотрим практический пример использования.

Представим, что мы пишем «пакет»: папку со множеством модулей, из которой часть функциональности экспортируется наружу (инструменты вроде NPM позволяют нам публиковать и распространять такие пакеты), а многие модули – просто вспомогательные, для внутреннего использования в других модулях пакета.

Структура файлов может быть такой:

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

Мы бы хотели сделать функциональность нашего пакета доступной через единую точку входа: «главный файл» `auth/index.js`. Чтобы можно было использовать её следующим образом:

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

Идея в том, что внешние разработчики, которые будут использовать наш пакет, не должны разбираться с его внутренней структурой, рыться в файлах внутри нашего пакета. Всё, что нужно, мы экспортируем в `auth/index.js`, а остальное скрываем от любопытных взглядов.

Так как нужная функциональность может быть разбросана по модулям нашего пакета, мы можем импортировать их в `auth/index.js` и тут же экспортировать наружу.

```
// 🍃 auth/index.js

// импортировать login/logout и тут же экспортировать
import {login, logout} from './helpers.js';
export {login, logout};

// импортировать экспорт по умолчанию как User и тут же
import User from './user.js';
export {User};
...
```

Теперь пользователи нашего пакета могут писать `import {login} from "auth/index.js"`.

Запись `export ... from ...` – это просто более короткий вариант такого импорта-экспорта:

```
// 🍃 auth/index.js

// импортировать login/logout и тут же экспортировать
export {login, logout} from './helpers.js';

// импортировать экспорт по умолчанию как User и тут же
export {default as User} from './user.js';
...
```

Реэкспорт экспорта по умолчанию

При реэкспорте экспорт по умолчанию нужно обрабатывать особым образом.

Например, у нас есть `user.js`, из которого мы хотим реэкспортировать класс `User`:

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

1. `export User from './user.js'` не будет работать. Казалось бы, что такого? Но возникнет синтаксическая ошибка!

   Чтобы реэкспортировать экспорт по умолчанию, мы должны написать `export {default as User}`, как в примере выше. Такая вот особенность синтаксиса.

2. `export * from './user.js'` реэкспортирует только именованные экспорты, исключая экспорт по умолчанию.

   Если мы хотим реэкспортировать и именованные экспорты и экспорт по умолчанию, то понадобятся две инструкции:

```
export * from './user.js'; // для реэкспорта именован
export {default} from './user.js'; // для реэкспорта
```

Такое особое поведение реэкспорта с экспортом по умолчанию – одна из причин того, почему некоторые разработчики их не любят.

Итого

Вот все варианты `export`, которые мы разобрали в этой и предыдущей главах.

Вы можете проверить себя, читая их и вспоминая, что они означают:

- Перед объявлением класса/функции/...:
  - `export [default] class/function/variable ...`
- Отдельный экспорт:
  - `export {x [as y], ...}`.
- Реэкспорт:
  - `export {x [as y], ...} from "module"`
  - `export * from "module"` (не реэкспортирует `export default`).
  - `export {default [as y]} from "module"` (реэкспортирует только `export default`).

Импорт:

- Именованные экспорты из модуля:
  - `import {x [as y], ...} from "module"`
- Импорт по умолчанию:
  - `import x from "module"`
  - `import {default as x} from "module"`
- Всё сразу:
  - `import * as obj from "module"`
- Только подключить модуль (его код запустится), но не присваивать его переменной:
  - `import "module"`

Мы можем поставить `import/export` в начало или в конец скрипта, это не имеет значения.

То есть, технически, такая запись вполне корректна:


```
sayHi();

// ...

import {sayHi} from './say.js'; // импорт в конце файла
```

На практике импорты, чаще всего, располагаются в начале файла. Но это только для большего удобства.

Обратите внимание, что инструкции import/export не работают внутри `{...}`.

Условный импорт, такой как ниже, работать не будет:

```
if (something) {
  import {sayHi} from "./say.js"; // Ошибка: импорт дол
}
```

...Но что, если нам в самом деле нужно импортировать что-либо в зависимости от условий? Или в определённое время? Например, загрузить модуль, только когда он станет нужен?

Мы рассмотрим динамические импорты в следующей главе.

Проводим курсы по JavaScript и фреймворкам.

Комментарии

перед тем как писать...