**Модули в ES6. import и export.**

### 1. Пример экспорта/импорта класса по дефолту

В index.html:

<script type="module" src="js/main.js"></script>

В main.js:

import **UserClass** from "./user.js"; // импорт дефолтного экспорта под именем UserVlass

const userBob = new UserClass("Bob", 28);

В user.js:

class User {

  constructor(name, age) {

    this.name = name;

    this.age = age;

  }

}

function printName(user) {

  console.log(`User name: ${user.name}`);

}

function printAge(user) {

  console.log(`User age: ${user.age}`);

}

export default User;

### 2. Пример экспорта/импорта класса не только по дефолту

В main.js:

import **UserClass** from "./user.js"; *// импорт дефолтного экспорта*

import { **printName, printAge** } from "./user.js";

const userBob = new UserClass("Bob", 28);

printName(userBob);

printAge(userBob);

В конце user.js:

export default User;

export **{ printName, printAge** };

### 3. При импорте недефолтных значений можем их переименовать

В main.js:

import { printName as **printUserNameFunc**, printAge } from "./user.js";

const userBob = new UserClass("Bob", 28);

printUserNameFunc(userBob);

### 4. Дефолтный и обычный импорт одной строкой

В main.js:

import UserClass, { printName as printUserNameFunc, printAge } from "./user.js";

### 5. Экспорт можно указать сразу при объявлении функции или класса

В user.js:

*// 3. Дефолтный экспорт сразу при определении класса (функции / переменной)*

export default class User {

constructor(name, age) {

this.name = name;

this.age = age;

}

}

// 4.1 Обычный экспорт сразу при определении функции / переменной / ...

export function printName(user) {

console.log(`User's name is: ${user.name}`);

}

### 6. Импорт всех экспортов через \*

В functions.js:

export function sum(x,y){

console.log(x+y);

}

export function sayHello (name){

console.log(`Hi ${name}! How are you?`);

}

В main.js:

import \* as **funcs** from './functions.js';

Далее функции используем как свойства объекта:

**funcs**.sum(5, 10);

**funcs**.sayHello('Bob');

**Учебник JS Модули**

Система модулей на уровне языка появилась в стандарте JavaScript в 2015 году и постепенно эволюционировала. На данный момент она поддерживается большинством браузеров и Node.js

Модуль – это просто файл. Один скрипт – это один модуль. Модули могут загружать друг друга и использовать директивы export и import, чтобы обмениваться функциональностью, вызывать функции одного модуля из другого:

* **export** отмечает переменные и функции, которые должны быть доступны вне текущего модуля.
* **import** позволяет импортировать функциональность из других модулей.

// sayHi.js

**export** function sayHi(user) {

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

}

// main.js

**import** {sayHi} from './sayHi.js';

alert(sayHi); // function...

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

Так как модули поддерживают ряд специальных ключевых слов, и у них есть ряд особенностей, то необходимо явно сказать браузеру, что скрипт является модулем, при помощи атрибута

<script type="module">.

## [Основные возможности модулей](https://learn.javascript.ru/modules-intro" \l "osnovnye-vozmozhnosti-moduley)

1) В модулях всегда используется режим use strict. Например, присваивание к необъявленной переменной вызовет ошибку.

<script type="module">

a = 5; // ошибка

</script>

2) Каждый модуль имеет свою собственную область видимости. Другими словами, переменные и функции, объявленные в модуле, не видны в других скриптах.

3) Если один и тот же модуль используется в нескольких местах, то его код выполнится только один раз, после чего экспортируемая функциональность передаётся всем импортёрам.

4) Объект import.meta содержит информацию о текущем модуле. Содержимое зависит от окружения. В браузере он содержит ссылку на скрипт или ссылку на текущую веб-страницу, если модуль встроен в HTML:

<script type="module">

alert(import.meta.url); // ссылка на html страницу для встроенного скрипта

</script>

5) В модуле на верхнем уровне this не определён (undefined). Сравним с не-модульными скриптами, там this – глобальный объект:

<script>

alert(this); // window

</script>

<script type="module">

alert(this); // undefined

</script>

## Особенности в браузерах

1) Модули являются отложенными (deferred). Это верно и для внешних и встроенных скриптов-модулей. Это означает:

* загрузка внешних модулей, таких как <script type="module" src="...">, не блокирует обработку HTML.
* модули, даже если загрузились быстро, ожидают полной загрузки HTML документа, и только затем выполняются.
* сохраняется относительный порядок скриптов: скрипты, которые идут раньше в документе, выполняются раньше.

Как побочный эффект, модули всегда видят полностью загруженную HTML-страницу, включая элементы под ними.

<script type="module">

alert(typeof button); // object: скрипт может 'видеть' кнопку под ним

// так как модули являются отложенными, то скрипт начнёт выполнятся только после полной загрузки страницы

</script>

Сравните с обычным скриптом ниже:

<script>

alert(typeof button); // Ошибка: кнопка не определена, скрипт не видит элементы под ним

// обычные скрипты запускаются сразу, не дожидаясь полной загрузки страницы

</script>

<button id="button">Кнопка</button>

Пожалуйста, обратите внимание: второй скрипт выполнится раньше, чем первый! Поэтому мы увидим сначала undefined, а потом object. При использовании модулей нам стоит иметь в виду, что HTML-страница будет показана браузером до того, как выполнятся модули и JavaScript-приложение будет готово к работе. Некоторые функции могут ещё не работать. Нам следует разместить «индикатор загрузки» или что-то ещё, чтобы не смутить этим посетителя.

2) Для не-модульных скриптов атрибут **async** работает только на **внешних скриптах**. Скрипты с ним запускаются сразу по готовности, они не ждут другие скрипты или HTML-документ. Для модулей атрибут async работает на любых скриптах.

Это очень полезно, когда модуль ни с чем не связан, например для счётчиков, рекламы, обработчиков событий.

<!-- загружаются зависимости (analytics.js) и скрипт запускается -->

<!-- модуль не ожидает загрузки документа или других тэгов <script> -->

<script async type="module">

import {counter} from './analytics.js';

counter.count();

</script>

3) Внешние скрипты с атрибутом type="module" имеют два отличия:

* Внешние скрипты с одинаковым атрибутом src запускаются только один раз:

<!-- скрипт my.js загрузится и будет выполнен только один раз -->

<script type="module" src="my.js"></script>

<script type="module" src="my.js"></script>

* Внешний скрипт, который загружается с другого домена, требует указания заголовков CORS. Другими словами, если модульный скрипт загружается с другого домена, то удалённый сервер должен установить заголовок Access-Control-Allow-Origin означающий, что загрузка скрипта разрешена.

<!-- another-site.com должен указать заголовок Access-Control-Allow-Origin -->

<!-- иначе, скрипт не выполнится -->

<script type="module" src="http://another-site.com/their.js"></script>

Это обеспечивает лучшую безопасность по умолчанию.

4) В **браузере** import должен содержать относительный или абсолютный путь к модулю. Модули без пути называются «голыми» (bare). Они не разрешены в import. Например, этот import неправильный:

import {sayHi} from 'sayHi'; // Ошибка, "голый" модуль

// путь должен быть, например './sayHi.js' или абсолютный

Другие окружения, например **Node.js**, допускают использование «голых» модулей, без путей, так как в них есть свои правила, как работать с такими модулями и где их искать.

5) **Старые браузеры** не понимают атрибут type="module". Скрипты с неизвестным атрибутом type просто игнорируются. Мы можем сделать для них «резервный» скрипт при помощи атрибута nomodule:

<script type="module">

alert("Работает в современных браузерах");

</script>

<script nomodule>

alert("Современные браузеры понимают оба атрибута - и type=module, и nomodule, поэтому **пропускают** этот тег script")

alert("Старые браузеры игнорируют скрипты с неизвестным атрибутом type=module, но выполняют этот.");

</script>

## [Инструменты сборки](https://learn.javascript.ru/modules-intro" \l "instrumenty-sborki)

В реальной жизни модули в браузерах редко используются в «сыром» виде. Обычно, мы объединяем модули вместе, используя специальный инструмент, например [Webpack](https://webpack.js.org/) и после выкладываем код на рабочий сервер.

Если мы используем инструменты сборки, то они объединяют модули вместе в один или несколько файлов, и заменяют import/export на свои вызовы. Поэтому итоговую сборку можно подключать и **без атрибута type="module**", как обычный скрипт:

<!-- Предположим, что мы собрали bundle.js, используя например утилиту Webpack -->

<script src="bundle.js"></script>

Импорт:

* Именованные экспорты из модуля: После экспорта класса/функции не принято ставить точку с зарятой. Мы можем поставить import/export в начало или в конец скрипта, это не имеет значения. На практике импорты, чаще всего, располагаются в начале файла.

import {x [as y], ...} from "module"

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');

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

// 📁 main.js

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

hi('John'); // Hello, John!

bye('John'); // Bye, John!

* Импорт по умолчанию: На практике модули встречаются в основном одного из двух типов:
  + Модуль, содержащий библиотеку или набор функций, как say.js выше.
  + 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/import default** («экспорт/импорт по умолчанию») для второго подхода.

import x from "module"

import {default as x} from "module"

* Импортируем все сразу:

import \* as obj from "module"

* Только подключить модуль (его код запустится), но не присваивать его переменной:

import "module"

## export

* Перед объявлением класса/функции/…

export [default] class/function/variable ...

В файле может быть не более одного export default

1. // 📁 user.js

export default class User { // просто добавьте "default"

constructor(name) {

this.name = name;

}

}

2// 📁 say.js

function sayHi(user) {

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

}

function sayBye(user) {

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

}

export {sayHi, sayBye}; // список экспортируемых переменных

3)// 📁 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');

* Отдельный экспорт:

export {x [as y], ...}.

* Реэкспорт:
  + export {x [as y], ...} from "module"

// 📁 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 экспортировать

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

Это более короткий вариант этого:

// 📁 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 экспортировать

import User from './user.js';

export {User};

* + export \* from "module" (не реэкспортирует export default).

реэкспортирует только именованные экспорты, исключая экспорт по умолчанию.

* + export {default [as y]} from "module" (реэкспортирует только export default).

реэкспорт с экспортом по умолчанию

Инструкции import/export **не работают внутри {...}.**

## Динамические импорты

Как мы можем импортировать модуль динамически, по запросу?

Выражение **import(module)** загружает модуль и возвращает **промис**, результатом которого становится объект модуля, содержащий все его экспорты. Использовать его мы можем динамически в любом месте кода, например, так:

let modulePath = prompt("Какой модуль загружать?");

import(modulePath)

.then(obj => <объект модуля>)

.catch(err => <ошибка загрузки, например если нет такого модуля>)

Или если внутри асинхронной функции, то можно

let module = await import(modulePath).

Например, если у нас есть такой модуль say.js:

// 📁 say.js

export function hi() {

alert(`Привет`);

}

export function bye() {

alert(`Пока`);

}

…То динамический импорт может выглядеть так

let {hi, bye} = await import('./say.js');

hi();

bye();

А если в say.js указан экспорт по умолчанию:

// 📁 say.js

export default function() {

alert("Module loaded (export default)!");

}

…То для доступа к нему нам следует взять свойство default объекта модуля:

let obj = await import('./say.js');

let say = **obj.default**;

// или, одной строкой:

let **{default: say}** = await import('./say.js');

say();

Хотя import() и выглядит похоже на вызов функции, на самом деле это специальный синтаксис, так же, как, например, super(). Так что мы не можем скопировать import в другую переменную или вызвать при помощи .call/apply. **Это не функция.**

## Передача параметра при подключении модуля

<http://code.mu/ru/javascript/book/supreme/modules/es/passing-parameters/>

Обернем код нашего модуля в функцию:

export default function(**data**) {

function func1() {

}

function func2() {

}

return {func1, func2};

}

Передадим данные параметром при вызове импортированной функции:

let **data** = 'abcde';

import func from './test.js';

let test = **func(data);**

Можем теперь вызвать функции нашего модуля:

test.func1();

test.func2();