**Лекция 16.**

**JavaScript: ES6 ­– Модули.**

[Модули 3](#_Toc519727046)

[Что такое модуль и зачем? 4](#_Toc519727047)

[Старые способы создания модулей 5](#_Toc519727048)

[Export 8](#_Toc519727049)

[Import 9](#_Toc519727050)

[Export default 10](#_Toc519727051)

[CommonJS 11](#_Toc519727052)

[Использование 12](#_Toc519727053)

[Дополнительные материалы 14](#_Toc519727054)

[Задание для закрепления материала 15](#_Toc519727055)

# Модули

Концепция модулей как способа организации JavaScript-кода существовала давно, и эта концепция может быть пугающей, но её понимание жизненно важно для разработчиков.

Когда приложение сложное и кода много – мы пытаемся разбить его на файлы. В каждом файле описываем какую-то часть, а в дальнейшем – собираем эти части воедино. Хорошие авторы делят свои книги на разделы и главы. Хорошие программисты делят свои программы на модули.

Хорошие модули обладают самодостаточными функциональными возможностями, поэтому они могут быть добавлены, перемещены или удалены по мере необходимости, не нарушив всю систему.

Модули в стандарте ECMAScript предоставляют удобные средства для этого.

Такие средства предлагались сообществом и ранее, например:

* AMD (Асинхронное определение модуля от англ. asynchronous module definition) – одна из самых древних систем организации модулей, требует лишь наличия клиентской библиотеки, к примеру, require.js, но поддерживается и серверными средствами. AMD – это подход к разработке на Javascript, при котором модули и их зависимости могут быть загружены асинхронно. Асинхронная загрузка модулей позволяет улучшить скорость загрузки веб-страницы в целом, так как модули загружаются одновременно с остальным контентом сайта.
* CommonJS – система модулей, встроенная в сервер Node.JS. Требует поддержки на клиентской и серверной стороне.
* UMD (Universal Module Definition) – система модулей, которая предложена в качестве универсальной. UMD-модули будут работать и в системе AMD и в CommonJS.

Все перечисленные выше системы требуют различных библиотек или систем сборки для использования.

Новый стандарт отличается от них прежде всего тем, что это – стандарт. А значит, со временем, будет поддерживаться браузерами без дополнительных утилит.

Сейчас поддержка браузерами хоть и наблюдается в последних версиях, но есть браузеры, где поддержки нет (IE и webview от Android например). Поэтому ES-модули используются в сочетании с системами сборки, такими как webpack, brunch и другими, при подключённом Babel.JS.

## Что такое модуль и зачем?

Модулем считается файл с кодом. В этом файле ключевым словом export помечаются переменные и функции, которые могут быть использованы снаружи. Другие модули могут подключать их через вызов import.

На самом деле у модулей есть много преимуществ. Наиболее важными являются следующие:

1. Удобная поддержка (Maintainability): По определению, модуль является самодостаточным. Хорошо спроектированный модуль призван уменьшить зависимости частей вашей кодовой базы насколько это возможно, чтобы она могла расти и совершенствоваться, не зависимо друг от друга. Обновить один модуль гораздо проще, когда он отделён от других частей кода. Возвращаясь к нашей книге, например, если вы захотите внести небольшое изменение в одну главу и это повлечёт за собой изменения какого-то другого раздела вашей книги, это будет кошмар. Поэтому главу нужно писать так, чтобы при внесении правок, не пришлось затрагивать другие главы.
2. Пространства имён (Namespacing): В JavaScript переменные, которые находятся за пределами функций верхнего уровня, считаются глобальными (каждый может получить к ним доступ). Поэтому очень распространено "Загрязнение пространства имён (namespace pollution)", где совершенно не связанный между собой код, связывают глобальные переменные. Совместное использование глобальных переменных в коде, который между собой не связан очень плохо в разработке. Модули позволяют избежать загрязнения глобального пространства имён, путём создания приватных пространств для наших переменных.
3. Повторное использование (Reusability): Давайте будем честными. Все мы копировали код в новые проекты, который писали раньше. Например, давайте представим, что вы скопировали в новый проект некоторые вспомогательные методы из предыдущего проекта. Хорошо, но если вы найдете наиболее хороший способ написать эту часть, вам придётся вспомнить все места где фигурировал этот код чтобы обновить его. Это безусловно огромная трата времени. Намного проще было бы написать модуль и использовать его повторно снова и снова.

## Старые способы создания модулей

Есть много способов интегрировать модули в свои программы. Давайте рассмотрим некоторые из них:

**Паттерн "Модуль"**

Паттерн "Модуль" используется для имитации концепции классов (так как изначально JavaScript не поддерживал классы), поэтому мы можем хранить публичные и приватные методы (переменные) внутри одного объекта. Это позволяет нам создать публичный API и предоставить возможность обращаться к публичным методам, в то время как приватные переменные и методы инкапсулированы в замыкании.

Есть несколько способов реализации паттерна "Модуль":

Пример 1: Анонимные замыкания

(function () {

// We keep these variables private inside this closure scope

var myGrades = [93, 95, 88, 0, 55, 91];

var average = function() {

var total = myGrades.reduce(function(accumulator, item) {

return accumulator + item

}, 0);

return 'Your average grade is ' + total / myGrades.length + '.';

}

var failing = function(){

var failingGrades = myGrades.filter(function(item) {

return item < 70;

});

return 'You failed ' + failingGrades.length + ' times.';

}

console.log(failing());

}());

Таким образом, у нашей анонимной функции есть своя область видимости или "замыкание" и мы можем сразу её выполнить. Такой способ позволяет нам скрыть переменные из родительской (глобальной) области видимости.

Что действительно хорошо в таком подходе, да это то, что вы можете использовать локальные переменные внутри этой функции и не бояться случайно, перезаписать глобальные переменные с таким же именем. Но доступ к глобальным переменным у вас по-прежнему остаётся.

Пример 2: Глобальный импорт

Другим популярным подходом, который используют библиотеки, такие как jQuery, является глобальный импорт. Он похож на замыкания, которые мы только что видели, только теперь мы передаём глобальные переменные в качестве параметров.

(function (globalVariable) {

// Keep this variables private inside this closure scope

var privateFunction = function() {

console.log('Shhhh, this is private!');

}

globalVariable.each = function(collection, iterator) {

if (Array.isArray(collection)) {

for (var i = 0; i < collection.length; i++) {

iterator(collection[i], i, collection);

}

} else {

for (var key in collection) {

iterator(collection[key], key, collection);

}

}

};

}(globalVariable));

В этом примере GlobalVariable единственная глобальная переменная. Преимущество такого подхода в том, что все глобальные переменные вы объявляете заранее, что делает ваш код прозрачным для остальных.

Пример 3: Объектный интерфейс

Ещё один подход при создании модулей заключается в использовании автономных, объектных интерфейсов, например так:

var myGradesCalculate = (function () {

// Keep this variable private inside this closure scope

var myGrades = [93, 95, 88, 0, 55, 91];

return {

average: function() {

var total = myGrades.reduce(function(accumulator, item) {

return accumulator + item;

}, 0);

return'Your average grade is ' + total / myGrades.length + '.';

},

failing: function() {

var failingGrades = myGrades.filter(function(item) {

return item < 70;

});

return 'You failed ' + failingGrades.length + ' times.';

}

}

})();

myGradesCalculate.failing(); // 'You failed 2 times.'

myGradesCalculate.average(); // 'Your average grade is 70.33333333333333.'

Как вы могли заметить, такой подход позволяет решать какие переменные (методы) мы хотим сделать приватными (например, myGrades), а какие публичными поместив их в возвращаемый объект (например, average и failing).

Пример 4: Паттерн "Раскрывающийся модуль"

Этот пример очень похож на предыдущий за исключением того, что все методы и переменные остаются приватными, пока их явно раскроют.

var myGradesCalculate = (function () {

// Keep this variable private inside this closure scope

var myGrades = [93, 95, 88, 0, 55, 91];

var average = function() {

var total = myGrades.reduce(function(accumulator, item) {

return accumulator + item;

}, 0);

return'Your average grade is ' + total / myGrades.length + '.';

};

var failing = function() {

var failingGrades = myGrades.filter(function(item) {

return item < 70;

});

return 'You failed ' + failingGrades.length + ' times.';

};

// Explicitly reveal public pointers to the private functions

// that we want to reveal publicly

return {

average: average,

failing: failing

}

})();

myGradesCalculate.failing(); // 'You failed 2 times.'

myGradesCalculate.average(); // 'Your average grade is 70.33333333333333.'

## Export

Ключевое слово export можно ставить:

* перед объявлением переменных, функций и классов.
* отдельно, при этом в фигурных скобках указывается, что именно экспортируется.

Например, так экспортируется переменная one:

// экспорт прямо перед объявлением

export let one = 1;

Можно написать export и отдельно от объявления:

let two = 2;

export {two};

При этом в фигурных скобках указываются одна или несколько экспортируемых переменных.

Для двух переменных будет так:

export {one, two};

При помощи ключевого слова as можно указать, что переменная one будет доступна снаружи (экспортирована) под именем once, а two – под именем twice:

export {one as once, two as twice};

Экспорт функций и классов выглядит так же:

export class User {

constructor(name) {

this.name = name;

}

};

export function sayHi() {

console.log("Hello!");

};

// отдельно от объявлений было бы так:

// export {User, sayHi}

Для экспорта обязательно нужно имя, в экспорте указываются именно имена, а не произвольные выражения. Без имени будет ошибка.

## Import

Другие модули могут подключать экспортированные значения при помощи ключевого слова import.

Синтаксис:

import {one, two} from "./nums";

Здесь:

* "./nums" – модуль, как правило это путь к файлу модуля.
* one, two – импортируемые переменные, которые должны быть обозначены в nums словом export.

В результате импорта появятся локальные переменные one, two, которые будут содержать значения соответствующих экспортов.

Например, при таком файле:

export let one = 1;

export let two = 2;

Модуль ниже выведет «1 and 2»:

import {one, two} from "./nums";

console.log( `${one} and ${two}` ); // 1 and 2

Импортировать можно и под другим именем, указав его в «as»:

// импорт one под именем item1, а two – под именем item2

import {one as item1, two as item2} from "./nums";

console.log( `${item1} and ${item2}` ); // 1 and 2

Можно импортировать все значения сразу в виде объекта вызовом import \* as obj, например:

import \* as numbers from "./nums";

// теперь экспортированные переменные - свойства numbers

alert( `${numbers.one} and ${numbers.two}` ); // 1 and 2

## Export default

Выше мы видели, что модуль может экспортировать выбранные переменные при помощи export.

Однако, как правило, код стараются организовать так, чтобы каждый модуль делал одну вещь. Иначе говоря, «один файл – одна сущность, которую он описывает». Например, файл user.js содержит class User, файл login.js – функцию login() для авторизации, и т.п.

При этом модули, разумеется, будут использовать друг друга. Например, login.js, скорее всего, будет импортировать класс User из модуля user.js.

Для такой ситуации, когда один модуль экспортирует одно значение, предусмотрено особое ключевое сочетание export default.

Если поставить после export слово default, то значение станет «экспортом по умолчанию».

Такое значение можно импортировать без фигурных скобок.

Например, файл user.js:

export default class User {

constructor(name) {

this.name = name;

}

};

…А в файле login.js:

import User from './user';

new User("Вася");

«Экспорт по умолчанию» – своего рода «синтаксический сахар». Можно было бы и без него, импортировать значение обычным образом через фигурные скобки {…}. Если бы в user.js не было default, то в login.js необходимо было бы указать фигурные скобки:

// если бы user.js содержал

// export class User { ... }

// …то при импорте User понадобились бы фигурные скобки:

import {User} from './user';

new User("Вася");

На практике этот «сахар» весьма приятен, так как позволяет легко видеть, какое именно значение экспортирует модуль, а также обойтись без лишних символов при импорте.

## CommonJS

Если вы раньше работали с Node.JS или использовали систему сборки в синтаксисе CommonJS, то вот соответствия.

Для экспорта по умолчанию вместо:

module.exports = VARIABLE;

Пишем:

export default VARIABLE;

А при импорте из такого модуля вместо:

const VARIABLE = require('./file');

Пишем:

import VARIABLE from './file';

Для экспорта нескольких значений из модуля, вместо:

exports.NAME = VARIABLE;

Пишем в фигурных скобках, что надо экспортировать и под каким именем (без as, если имя совпадает):

export {VARIABLE as NAME};

При импорте – также фигурные скобки:

import {NAME} from './file';

## Использование

Современный стандарт ECMAScript описывает, как импортировать и экспортировать значения из модулей, но он ничего не говорит о том, как эти модули искать, загружать и т.п.

Такие механизмы предлагались в процессе создания стандарта, но были убраны по причине недостаточной проработанности. Возможно, они появятся в будущем.

Сейчас используются системы сборки, как правило, в сочетании с Babel.JS.

Система сборки обрабатывает скрипты, находит в них import/export и заменяет их на свои внутренние JavaScript-вызовы. При этом, как правило, много файлов-модулей объединяются в один или несколько скриптов, смотря, как указано в конфигурации сборки.

# Дополнительные материалы

* <https://learn.javascript.ru/closures-module>
* <https://tuhub.ru/posts/javascript-moduli-rukovodstvo-dlya-nachinayushhih>
* <https://learn.javascript.ru/modules>

# Задание для закрепления материала

Калькулятор продолжается:

Переписать функционал используя export и import, добиться модульности и независимости каждого отдельно взятого модуля