**Модули через замыкания**

Приём программирования «модуль» имеет громадное количество вариаций. Он немного похож на счётчик, который мы рассматривали ранее, использует аналогичный приём, но на уровне выше.

Его цель – скрыть внутренние детали реализации скрипта. В том числе: временные переменные, константы, вспомогательные мини-функции и т.п.

**[Зачем нужен модуль?](http://learn.javascript.ru/closures-module" \l "zachem-nuzhen-modul)**

Допустим, мы хотим разработать скрипт, который делает что-то полезное на странице.

Умея работать со страницей, мы могли бы сделать много чего, но так как пока этого не было (скоро научимся), то пусть скрипт просто выводит сообщение:

Файл hello.js

// глобальная переменная нашего скрипта

var message = "Привет";

// функция для вывода этой переменной

function showMessage() {

alert( message );

}

// выводим сообщение

showMessage();

У этого скрипта есть свои внутренние переменные и функции.

В данном случае это message и showMessage.

Предположим, что мы хотели бы распространять этот скрипт в виде библиотеки. Каждый, кто хочет, чтобы посетителям выдавалось «Привет» – может просто подключить этот скрипт. Достаточно скачать и подключить, например, как внешний файл hello.js – и готово.

**Если подключить подобный скрипт к странице «как есть», то возможен конфликт с переменными, которые она использует.**

То есть, при подключении к такой странице он её «сломает»:

<script>

var message = "Пожалуйста, нажмите на кнопку";

</script>

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

<button>Кнопка</button>

<script>

// ожидается сообщение из переменной выше...

alert( message ); // но на самом деле будет выведено "Привет"

</script>

[открыть в песочнице](https://plnkr.co/edit/gEnNSIQIJ9XLmHiVty3w?p=preview)

Автор страницы ожидает, что библиотека "hello.js" просто отработает, без побочных эффектов. А она вместе с этим переопределила message в "Привет".

Если же убрать скрипт hello.js, то страница будет выводить правильное сообщение.

Зная внутреннее устройство hello.js нам, конечно, понятно, что проблема возникла потому, что переменная message из скрипта hello.js перезаписала объявленную на странице.

## [Приём проектирования «Модуль»](http://learn.javascript.ru/closures-module" \l "priyom-proektirovaniya-modul)

Чтобы проблемы не было, всего-то нужно, чтобы у скрипта была своя собственная область видимости, чтобы его переменные не попали на страницу.

Для этого мы завернём всё его содержимое в функцию, которую тут же запустим.

Файл hello.js, оформленный как модуль:

(function() {

// глобальная переменная нашего скрипта

var message = "Привет";

// функция для вывода этой переменной

function showMessage() {

alert( message );

}

// выводим сообщение

showMessage();

}());

[открыть в песочнице](https://plnkr.co/edit/Lj0Qwn6cqaa0wDZHBW6c?p=preview)

Этот скрипт при подключении к той же странице будет работать корректно.

Будет выводиться «Привет», а затем «Пожалуйста, нажмите на кнопку».

### [Зачем скобки вокруг функции?](http://learn.javascript.ru/closures-module" \l "zachem-skobki-vokrug-funktsii)

В примере выше объявление модуля выглядит так:

(function() {

alert( "объявляем локальные переменные, функции, работаем" );

// ...

}());

В начале и в конце стоят скобки, так как иначе была бы ошибка.

Вот, для сравнения, неверный вариант:

function() {

// будет ошибка

}();

Ошибка при его запуске произойдет потому, что браузер, видя ключевое слово function в основном потоке кода, попытается прочитать Function Declaration, а здесь имени нет.

Впрочем, даже если имя поставить, то работать тоже не будет:

function work() {

// ...

}(); // syntax error

**Дело в том, что «на месте» разрешено вызывать только Function Expression.**

Общее правило таково:

* Если браузер видит function в основном потоке кода – он считает, что это Function Declaration.
* Если же function идёт в составе более сложного выражения, то он считает, что это Function Expression.

Для этого и нужны скобки – показать, что у нас Function Expression, который по правилам JavaScript можно вызвать «на месте».

Можно показать это другим способом, например поставив перед функцией оператор:

+function() {

alert('Вызов на месте');

}();

!function() {

alert('Так тоже будет работать');

}();

**[Экспорт значения](http://learn.javascript.ru/closures-module" \l "eksport-znacheniya)**

Приём «модуль» используется почти во всех современных библиотеках.

Ведь что такое библиотека? Это полезные функции, ради которых её подключают, плюс временные переменные и вспомогательные функции, которые библиотека использует внутри себя.

Посмотрим, к примеру, на библиотеку [Lodash](http://lodash.com/), хотя могли бы и [jQuery](http://jquery.com/), там почти то же самое.

Если её подключить, то появится специальная переменная lodash (короткое имя \_), которую можно использовать как функцию, и кроме того в неё записаны различные полезные свойства, например:

* \_.defaults(src, dst1, dst2...) – копирует в объект src те свойства из объектов dst1, dst2 и других, которых там нет.
* \_.cloneDeep(obj) – делает глубокое копирование объекта obj, создавая полностью независимый клон.
* \_.size(obj) – возвращает количество свойств в объекте, полиморфная функция: можно передать массив или даже 1 значение.

Есть и много других функций, подробнее описанных в [документации](https://lodash.com/docs).

Пример использования:

<p>Подключим библиотеку</p>

<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"></script>

<p>Функция <code>\_.defaults()</code> добавляет отсутствующие свойства.</p>

<script>

var user = {

name: 'Вася'

};

\_.defaults(user, {

name: 'Не указано',

employer: 'Не указан'

});

alert( user.name ); // Вася

alert( user.employer ); // Не указан

alert( \_.size(user) ); // 2

</script>

Здесь нам не важно, какие функции или методы библиотеки используются, нас интересует именно как описана эта библиотека, как в ней применяется приём «модуль».

Вот примерная выдержка из исходного файла:

;(function() {

// lodash - основная функция для библиотеки

function lodash(value) {

// ...

}

// вспомогательная переменная

var version = '2.4.1';

// ... другие вспомогательные переменные и функции

// код функции size, пока что доступен только внутри

function size(collection) {

return Object.keys(collection).length;

}

// присвоим в lodash size и другие функции, которые нужно вынести из модуля

lodash.size = size

// lodash.defaults = ...

// lodash.cloneDeep = ...

// "экспортировать" lodash наружу из модуля

window.\_ = lodash; // в оригинальном коде здесь сложнее, но смысл тот же

}());

Внутри внешней функции:

1. Происходит что угодно, объявляются свои локальные переменные, функции.
2. В window выносится то, что нужно снаружи.

Технически, мы могли бы вынести в window не только lodash, но и вообще все объекты и функции. На практике, как раз наоборот, всё прячут внутри модуля, глобальную область во избежание конфликтов хранят максимально чистой.

**Зачем точка с запятой в начале?**

В начале кода выше находится точка с запятой ; – это не опечатка, а особая «защита от дураков».

Если получится, что несколько JS-файлов объединены в один (и, скорее всего, сжаты минификатором, но это не важно), и программист забыл поставить точку с запятой, то будет ошибка.

Например, первый файл a.js:

var a = 5

Второй файл lib.js:

(function() {

// без точки с запятой в начале

})()

После объединения в один файл:

var a = 5

// библиотека

(function() {

// ...

})();

При запуске будет ошибка, потому что интерпретатор перед скобкой сам не вставит точку с запятой. Он просто поймёт код как var a = 5(function ...), то есть пытается вызвать число 5 как функцию.

Таковы правила языка, и поэтому рекомендуется явно ставить точку с запятой. В данном случае автор lodash ставит ; перед функцией, чтобы предупредить эту ошибку.

## [Экспорт через return](http://learn.javascript.ru/closures-module" \l "eksport-cherez-return)

Можно оформить модуль и чуть по-другому, например передать значение через return:

var lodash = (function() {

var version;

function assignDefaults() { ... }

return {

defaults: function() { }

}

})();

Здесь, кстати, скобки вокруг внешней function() { ... } не обязательны, ведь функция и так объявлена внутри выражения присваивания, а значит – является Function Expression.

Тем не менее, лучше их ставить, для улучшения читаемости кода, чтобы было сразу видно, что это не простое присвоение функции.

## [Итого](http://learn.javascript.ru/closures-module" \l "itogo)

Модуль при помощи замыканий – это оборачивание пакета функционала в единую внешнюю функцию, которая тут же выполняется.

Все функции модуля будут иметь доступ к другим переменным и внутренним функциям этого же модуля через замыкание.

Например, defaults из примера выше имеет доступ к assignDefaults.

Но снаружи программист, использующий модуль, может обращаться напрямую только к тем переменным и функциям, которые экспортированы. Благодаря этому будут скрыты внутренние аспекты реализации, которые нужны только разработчику модуля.

Можно придумать и много других вариаций такого подхода. В конце концов, «модуль» – это всего лишь функция-обёртка для скрытия переменных.