

Современный учебник JavaScript

© Илья Кантор

Сборка от 23 июля 2013 для чтения с устройств

Внимание, эта сборка может быть устаревшей и не соответствовать текущему тексту.

Актуальный онлайн-учебник, с интерактивными примерами, доступен по адресу <http://learn.javascript.ru>.

Вопросы по JavaScript можно задавать в комментариях на сайте или на форуме javascript.ru/forum.

Вопросы по сборке, предложения по её улучшению – можно писать мне, по адресу iliakan@javascript.ru.

Глава: Функции и замыкания

В файле находится только одна глава учебника. Это сделано в целях уменьшения размера файла, для удобного чтения с устройств.

Содержание

Функция - это значение

Функция — это значение

Копирование функций

Function Declaration и Function Expression

Объявление Function Declaration

Время создания Function Declaration

Объявление Function Expression

Функция с вызовом «на месте»

Итого

Именованные функциональные выражения

Глобальный объект

- Глобальный объект

- Порядок инициализации

- Итого

Замыкания, функции изнутри

- Лексическое окружение

 - Пример

- Доступ ко внешним переменным

- Вложенные функции

- Управление памятью

- [[Scope]] для new Function

- Итого

Хранение данных в замыкании, модули

- Данные для счётчика

 - Объект счётчика

 - Объект счётчика + функция

- Приём проектирования «Модуль»

- Задачи на понимание замыканий

Статические переменные

- Использование замыкания

- Запись в функцию

Конструкция "with"

- Пример

- Изменения переменной

- Почему отказались от with?

 - Замена with

- Итого

Решения задач

Функция - это значение

В этом разделе мы познакомимся с важными особенностями функций в JavaScript, а также с тремя способами объявить функцию.

Функция — это значение

В JavaScript функция является значением, таким же как строка или число.

Объявление создает функцию и записывает ссылку на неё в переменную.

Как и любое значение, функцию можно вывести, вот так:

```
1 function sayHi() {  
2   alert('Привет');  
3 }  
4  
5 alert(sayHi); // выведет код функции
```

Здесь выводится не результат работы функции `sayHi()` (кстати, чему он равен?.. правильно, `undefined`, т.к. нет `return`), а сама функция, т.е. ее код.

Функция — не просто значение, это объект.

В него можно даже записать свойство:

```
1 function sayHi() { }  
2  
3 sayHi.test = 5;  
4  
5 alert(sayHi.test); // 5
```

Используется эта возможность достаточно редко.

Копирование функций

Функцию можно скопировать в другую переменную.

```
1 function sayHi(person) {  
2   alert('Привет, ' + person);  
3 }
```

```
4  
5 var func = sayHi;  
6  
7 alert(func); // выведет код функции
```

При этом, так как функция — это объект, то и копируется она «по ссылке».

То есть, сама функция лежит где-то в памяти, а переменная содержит «адрес», где она находится. При присваивании `func = sayHi` копируется этот адрес, обе переменные начинают указывать на одно и то же «место в памяти», где находится функция.

После копирования ее можно вызывать:

```
1 function sayHi(person) {  
2   alert('Привет, ' + person);  
3 }  
4  
5 var func = sayHi;  
6  
7 func('Вася'); // выведет 'Привет, Вася'  
8  
9 sayHi('Маша'); // и так по-прежнему работает: 'Привет,  
   Маша'
```

Заметим ещё раз, что обращение вида `alert(func)` отличается от `alert(func())`. В первом случае выводится функция (её код), во втором — эта функция вызывается, и выведен будет уже её результат.

Function Declaration и Function Expression

В этом разделе мы познакомимся с важными особенностями функций в JavaScript, а также с тремя способами объявить функцию.

Объявление Function Declaration

Объявление функции, о котором мы говорили все время до этого, называется в спецификации языка `Function Declaration`.

Устоявшегося русского перевода нет, также такой синтаксис называют «стандартным» или «обычным» объявлением функции. Чуть дальше мы посмотрим [другие варианты](#) создания функций.

Позвольте еще раз выделить основной смысл объявления

Function Declaration:



При объявлении функции создаётся переменная со значением-функцией

Другими словами, объявление

`function func(параметры) { код }` говорит интерпретатору: «создай переменную `func`, и положи туда функцию с указанными параметрами и кодом».

При этом `func` — на самом деле не «имя функции». Оно совсем никак не привязано к функции!

Это название переменной, в которую будет помещена функция, и из которой она может быть затем удалена, скопирована в другую переменную, и в результате её как `func` вызвать будет нельзя:

```
1 function func() { alert(1); }
2
3 var g = func; // скопировали
4
5 func = null; // поменяли значение
6
7 g(); // работает, теперь функция в g, а в func
  ничего нет
8 func(); // вызываем null()? ошибка!
```

В частности, невозможно иметь функцию и переменную с одинаковым именем:

```
1 // Нонсенс!
2 function f() { } // объявить переменную f и записать в нее
  функцию
3 var f = 5; // объявить переменную f (а она уже объявлена)
  и присвоить 5
```

```
4  
5 alert(f); // результат: 5
```

В примере выше переменная `f` в первой строке получает значение — функцию, а во второй — число 5. В итоге мы получим переменную со значением `f=5`.

Время создания Function Declaration

Функции, объявленные как `Function Declaration`, создаются интерпретатором до выполнения кода.

Перед тем, как выполнять первую строку, интерпретатор сканирует код, ищет в нём `Function Declaration` и обрабатывает их.

Поэтому их можно вызвать до объявления, например:

```
1 sayHi("Вася");  
2  
3 function sayHi(name) {  
4   alert("Привет, " + name);  
5 }
```

Этот код будет работать, т.к. объявление `function sayHi` обрабатывается и функция создаётся до выполнения первой строчки кода.



Условно объявить функцию через `Function Declaration` нельзя

Попробуем, в зависимости от условия, объявить функцию по-разному:

```
1 var age = 20;  
2  
3 if (age >= 18) {  
4   function sayHi() { alert('Прошу вас!'); }  
5 } else {  
6   function sayHi() { alert('До 18 нельзя'); }  
7 }
```

```
8  
9 sayHi();
```

Какой вызов сработает?

Чтобы это понять — вспомним, как работают функции.

1. **Function Declaration** обрабатываются до выполнения первой строки кода.

Браузер сканирует код и создает из таких объявлений функции. При этом второе объявление перезапишет первое.

2. Дальше, во время выполнения объявления игнорируются (они уже обработаны), как если бы код был таким:

```
01 function sayHi() { alert('Прощу вас!'); }  
02 function sayHi() { alert('До 18 нельзя'); }  
03  
04 var age = 20;  
05  
06 if (age >= 18) {  
07  
08 } else {  
09  
10 }  
11  
12 sayHi();
```

Как видно, конструкция `if` здесь ни на что не влияет. По-разному объявить функцию, в зависимости от условия, не получилось.

P.S. Это — правильное с точки зрения спецификации поведение. На момент написания этого раздела ему следуют все браузеры, кроме Firefox.

Объявление **Function Expression**

Существует альтернативный синтаксис для создания функций, который решает проблемы с «условным» объявлением.

Функцию можно создать и присвоить переменной как самое обычное значение.

Такое объявление называется Function Expression и выглядит так:

```
1 | var f = function(параметры) {  
2 |     // тело функции  
3 | };
```

Например:

```
1 | var sayHi = function(person) {  
2 |     alert("Привет, " + person);  
3 | };  
4 |  
5 | sayHi('Вася');
```

Такую функцию можно объявить в любом выражении, там где допустимо обычное значение.

Например, вполне допустимо такое объявление массива:

```
1 | var arr = [1, 2, function(a) { alert(a) }, 3, 4];  
2 |  
3 | var fun = arr[2];  
4 | fun(1); // 1
```

В отличие от объявлений Function Declaration, которые создаются заранее, до выполнения кода, объявления Function Expression создают функцию, когда до них доходит выполнение.

Поэтому и пользоваться ими можно только после объявления.

```
1 | sayHi(); // <-- работать не будет, функции еще нет  
2 |  
3 | var sayHi = function() { alert(1) };
```

... А вот так — будет:

```
1 | var sayHi = function() { alert(1) };  
2 |  
3 | sayHi(); // 1
```

Благодаря этому свойству Function Expression можно (и даже нужно)

использовать для условного объявления функции:

```
01 var age = prompt('Сколько вам лет?');
02 var sayHi;
03
04 if (age >= 18) {
05     sayHi = function() { alert('Вход разрешен'); }
06 } else {
07     sayHi = function() { alert('Извините, вы слишком
08     молоды'); }
09 }
10 sayHi(); // запустит ту функцию, которая присвоена в if
```

Функция с вызовом «на месте»

Представим на минуту, что мы создали скрипт, который делает нечто восхитительно-ошеломительное со страницей. Для этого он, конечно же, объявляет свои временные переменные и функции.

Наш скрипт настолько замечательный, что мы хотим дать его другим людям.

В этом случае мы бы хотели, чтобы любой мог вставить скрипт на страницу, и временные переменные и функции скрипта не конфликтовали с теми, которые на ней используются. Например, наш скрипт задаёт переменные `a`, `b`, и если на странице они уже используются — будет конфликт.

Чтобы его не было, нашему скрипту нужна своя отдельная область видимости, в которой он будет работать.

Для этого используют функции с вызовом «на месте». Такая функция объявляется — и тут же вызывается, вот так:

```
1 (function() {
2
3     var a = 1 , b = 2; // переменные для нашего скрипта
4
5     // код скрипта
6
7 })();
```

Теперь внутренние переменные скрипта стали локальными. Даже если в другом месте страницы объявлена своя переменная `a` — проблем не будет. Наш скрипт теперь имеет свою, изолированную область видимости.

Эта практика широко используется в библиотеках JavaScript, чтобы временные переменные и функции, которые используются при инициализации, не конфликтовали с внешним кодом.

У функции, которая объявлена таким образом, и тут же вызвана, нет имени. Она никуда не сохраняется, и поэтому не может быть вызвана второй раз. Но в данном случае это и не нужно, ведь задача такой функции обёртки — создать отдельную область видимости для скрипта, что она и сделала.



Зачем скобки вокруг функции?

В примерах выше вокруг `function() { ... }` находятся скобки. Если записать без них — такой код вызовет ошибку:

```
1 function() {  
2   // syntax error  
3 }();
```

Эта ошибка произойдет потому, что браузер, видя ключевое слово `function` в основном потоке кода, попытается прочитать `Function Declaration`, а здесь даже имени нет.

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

```
1 function work() {  
2   // ...  
3 }() // 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 так:

```
+function() { ... }()
```

Или так:

```
!function() { ... }()
```

Главное, чтобы интерпретатор понял, что это `Function Expression`, тогда он позволит вызвать функцию «на месте».

Скобки не нужны, если это и так `Function Expression`, например в таком вызове:

```
1 | var res = function(a,b) { return a+b }(2,2);  
2 | alert(res); // 4
```

Функция здесь создаётся как часть выражения присваивания, поэтому является `Function Expression` и может быть вызвана «на месте».



При вызове «на месте» лучше ставить скобки и для `Expression`

Технически, если функция и так задана как `Function Expression`, то она может быть вызвана «на месте» без скобок:

```
var result = function(a,b) {  
    return a*b;  
}(2,3);
```

Но из соображений стиля и читаемости скобки вокруг `function` рекомендуется ставить:

```
var result = (function(a,b) {  
    return a*b;  
})(2,3);
```

Тогда сразу видно, что в `result` записывается не сама функция (`result = function...`), а идёт «вызов на месте». При чтении такого кода меньше ошибок.



Создание через `new Function`

Существует еще один способ создания функции, который использует прямой вызов встроенного конструктора `Function`. Он принимает список аргументов и тело в виде строки, и генерирует функцию по ним:

```
1 var sayHi = new Function('name', ' alert("Привет,  
2     "+name) ');  
3 sayHi("Вася");
```

Этот способ используется очень редко.



Предпочитайте `Function Declaration`

В коде неопытных разработчиков зачастую можно увидеть много функций, объявленных как `Function Expression`:

```
... код ...  
var f = function() { ... }  
...
```

То же самое с `Function Declaration` будет короче и читабельнее:

```
... код ...  
function f() { ... }  
...
```

Дополнительный бонус - такие функции можно вызывать до того, как они объявлены.

Используйте Function Expression только там, где это действительно нужно. Например, для объявления функции в зависимости от условий.

Итого

Функции в JavaScript являются значениями. Их можно присваивать, передавать, создавать в любом месте кода.

- Если функция объявлена в *основном потоке кода*, то это Function Declaration.
- Если функция создана как часть выражения, то это Function Expression.

Между этими двумя основными способами создания функций есть следующие различия:

	Function Declaration	Function Expression
Время создания	До выполнения первой строки кода.	Когда управление достигает строки с функцией.
Можно вызвать до объявления	Да (т.к. создается заранее)	Нет
Можно объявить в if	Нет (т.к. создается заранее)	Да
Можно вызывать «на месте»	Нет (ограничение синтаксиса JavaScript)	Да

По возможности, рекомендуется использовать Function Declaration, если нет особой причины для Function Expression.

Именованные функциональные

выражения

Обычно у функций в JavaScript нет имени. Как мы говорили ранее, функция является всего лишь значением, которое присваивается переменной.

То, что обычно называют «именем функции» — на самом деле, является именем переменной. К самой функции это «имя» никак не привязано. Если функцию переместить в другую переменную — она сменит «имя»:

```
1 function f() { alert(1); };
2 g = f;
3 f = 0;
4
5 g(); // сменили имя f на g!
```

Однако, есть в JavaScript расширение `Function Expression`, позволяющее указать имя, действительно привязанное к функции. Оно называется `Named Function Expression (NFE)` или, по-русски, *именованное функциональное выражение*.

Простейший пример NFE выглядит так:

```
var f = function sayHi(...) { /* тело функции */ };
```

Проще говоря, NFE — это `Function Expression` с дополнительным именем (в примере выше `sayHi`).

Что же это за имя, которое идёт в дополнение к переменной `f`, и зачем оно?

Имя функционального выражения (`sayHi`) имеет особый смысл. Оно доступно только изнутри самой функции.

Это ограничение видимости входит в стандарт JavaScript и поддерживается всеми браузерами, кроме IE<9.

Например:

```
1 var f = function sayHi(name) {  
2   alert(sayHi); // изнутри функции - видно (выведет код  
   функции)  
3 };  
4  
5 alert(sayHi); // снаружи - не видно (ошибка: undefined  
   variable 'sayHi')
```

Как правило, оно используется для единственной цели — позволить функции вызвать саму себя.



Устаревшее специальное значение `arguments.callee`

Если вы работали с JavaScript, то, возможно, знаете, что для этой цели также служило специальное значение `arguments.callee`.

Если это название вам ни о чём не говорит — всё в порядке, читайте дальше, мы обязательно обсудим его [в отдельной главе](#).

Если же вы в курсе, то стоит иметь в виду, что оно официально исключено из современного стандарта. А NFE — это наше настоящее.

Такое имя жёстко привязано к функции.

Оно работает всегда, даже если переменная, в которой изначально содержалась функция, была перезаписана, а сама функция — перемещена в другое место.

Например, объявим функцию-факториал из задачи [Вычислить факториал](#) как NFE:

```
var f = function factorial(n) {  
  return n ? n*factorial(n-1) : 1;  
};
```

Почему мы решили не использовать `Function Declaration`, а присвоить её в переменную `f`? Здесь — для примера, а вообще, причин бывает множество, скажем, условное создание функции внутри `if`, при котором `Function Declaration` [не работает](#).

После создания функции, даже если переменная `f` перезаписана,

функция сможет обращаться к самой себе через имя `factorial`:

```
1 var f = function factorial(n) {
2   return n ? n*factorial(n-1) : 1;
3 };
4
5 var g = f; // скопировали ссылку на функцию-факториал в g
6
7 f = function() { return 0; }; // перезаписали f
8
9 alert( g(5) ); // 120, работает!
```

Сравним это с обычным Function Expression:

```
1 var f = function(n) {
2   return n ? n*f(n-1) : 1;
3 };
4
5 var g = f;
6
7 f = null;
8
9 alert( g(5) ); // ошибка, т.к. g попытается вызвать f(n-1), а f перезаписана
```



В браузере IE8 и ниже создаются две функции

Как мы говорили выше, в браузере IE до 9 версии имя NFE видно везде, что является ошибкой с точки зрения стандарта.

..Но на самом деле ситуация еще забавнее. Старый IE создаёт в таких случаях целых две функции: одна записывается в переменную `f`, а вторая — в переменную `factorial`.

Например:

```
1 var f = function factorial(n) { /*...*/ };
2
3 // в IE8- false
4 // в остальных браузерах ошибка, т.к. имя factorial
  не видно
5 alert(f === factorial);
```

Все остальные браузеры полностью поддерживают именованные

функциональные выражения.



Задача: Проверка на NFE

Каков будет результат выполнения кода?

Важность: 5

```
function g() { return 1; }
```

```
alert(g);
```

А такого? Будет ли разница, если да — почему?

```
(function g() { return 1; });
```

```
alert(g);
```

[Решение задачи "Проверка на NFE" »»](#)



Задача: Перезапись NFE по имени

Что будет при повторном вызове функции?

Важность: 5

```
1 var f = function sayHi(n) {  
2   alert(n);  
3   sayHi = 0; // перезаписываем имя функции  
4 };  
5  
6 f(1); // что выведет?  
7 f(2); // а теперь что выведет?
```

Разумеется, в жизни мы не делаем такого, но разобрать ситуацию важно для общего понимания функций.

[Решение задачи "Перезапись NFE по имени" »»](#)

Глобальный объект

Механизм работы функций и переменных в JavaScript очень отличается от большинства языков.

Чтобы его понять, мы в этой главе рассмотрим переменные и функции в

глобальной области. А в следующей — пойдём дальше.

Глобальный объект

Глобальными называют переменные и функции, которые не находятся внутри какой-то функции. То есть, иными словами, если переменная или функция не находятся внутри конструкции `function`, то они — «глобальные».

В JavaScript все глобальные переменные и функции являются свойствами специального объекта, который называется «глобальный объект» (`global object`).

В браузере этот объект явно доступен под именем `window`. Объект `window` одновременно является глобальным объектом и содержит ряд свойств и методов для работы с окном браузера, но нас здесь интересует только его роль как глобального объекта.

В других окружениях, например Node.js, глобальный объект может быть недоступен в явном виде, но суть происходящего от этого не изменяется, поэтому далее для обозначения глобального объекта мы будем использовать `"window"`.

Присваивая или читая глобальную переменную, мы, фактически, работаем со свойствами `window`.

Например:

```
1 | var a = 5;    // объявление var создаёт свойство window.a
2 | alert(window.a); // 5
```

Создать переменную можно и явным присваиванием в `window`:

```
1 | window.a = 5;
2 | alert(a); // 5
```

Порядок инициализации

Выполнение скрипта происходит в две фазы:

1. На первой фазе происходит инициализация, подготовка к запуску.

Во время инициализации скрипт сканируется на предмет объявления функций вида **Function Declaration**, а затем — на предмет объявления переменных `var`. Каждое такое объявление добавляется в `window`.

Функции, объявленные как **Function Declaration, создаются сразу работающими, а переменные — равными `undefined`.**

2. На второй фазе — собственно, выполнение.

Присваивание (`=`) значений переменных происходит на второй фазе, когда поток выполнения доходит до соответствующей строки кода.

В начале кода ниже указано содержание глобального объекта на момент окончания инициализации:

```
1 // По окончании инициализации, до выполнения кода:
2 // window = { f: function, a: undefined, g: undefined }
3
4 var a = 5; // при инициализации даёт: window.a=undefined
5
6 function f(arg) { /*...*/ } // при инициализации даёт:
  window.f = function
7
8 var g = function(arg) { /*...*/ }; // при инициализации
  даёт: window.g = undefined
```

Кстати, тот факт, что к началу выполнения кода переменные и функции уже содержатся в `window`, можно легко проверить:

```
1 alert("a" in window); // true, т.к. есть свойство
  window.a
2 alert(a); // равно undefined, присваивание будет
  выполнено далее
3 alert(f); // function ..., готовая к выполнению функция
4 alert(g); // undefined, т.к. это переменная, а не
  Function Declaration
5
6 var a = 5;
7 function f() { /*...*/ }
8 var g = function() { /*...*/ };
```



Присвоение переменной без объявления

В старом стандарте JavaScript переменную можно было создать и

без объявления `var`:

```
1 | a = 5;  
2 |  
3 | alert(a); // 5
```

Такое присвоение, как и `var a = 5`, создает свойство `window.a = 5`. Отличие от `var a = 5` — в том, что переменная будет создана не на этапе входа в область видимости, а в момент присвоения.

Сравните два кода ниже.

Первый выведет `undefined`, так как переменная была добавлена в `window` на фазе инициализации:

```
1 | alert(a); // undefined  
2 |  
3 | var a = 5;
```

Второй код выведет ошибку, так как переменной ещё не существует:

```
1 | alert(a); // error, a is not defined  
2 |  
3 | a = 5;
```

Вообще, рекомендуется всегда объявлять переменные через `var`.

В современном стандарте присваивание без `var` вызовет ошибку:

```
1 | 'use strict';  
2 | a = 5; // error, a is not defined
```



Конструкции `for`, `if...` не влияют на видимость переменных

Фигурные скобки, которые используются в `for`, `while`, `if`, в отличие от объявлений функции, имеют «декоративный»

характер.

В JavaScript нет разницы между объявлением вне блока:

```
1 | var i;  
2 | {  
3 |   i = 5;  
4 | }
```

...И внутри него:

```
1 | i = 5;  
2 | {  
3 |   var i;  
4 | }
```

Также нет разницы между объявлением в цикле и вне его:

```
1 | for (var i=0; i<5; i++) { }
```

Идентичный по функциональности код:

```
1 | var i;  
2 | for (i=0; i<5; i++) { }
```

В обоих случаях переменная будет создана до выполнения цикла, на стадии инициализации, и ее значение будет сохранено после окончания цикла.



Не важно, где и сколько раз объявлена переменная

Объявлений `var` может быть сколько угодно:

```
1 | var i = 10;  
2 |  
3 | for (var i=0; i<20; i++) {  
4 |   ...  
5 | }  
6 |  
7 | var i = 5;
```

Все `var` будут обработаны один раз, на фазе инициализации.

На фазе исполнения объявления `var` будут проигнорированы: они уже были обработаны. Зато будут выполнены присваивания.



Ошибки при работе с `window` в IE8-

В старых IE есть две забавные ошибки при работе с переменными в `window`:

1. Переопределение переменной, у которой такое же имя, как и `id` элемента, приведет к ошибке:

```
1 <div id="a">...</div>
2 <script>
3   a = 5;    // ошибка в IE<9! Правильно будет
   "var a = 5"
4   alert(a); // никогда не сработает
5 </script>
```

А если сделать через `var`, то всё будет хорошо.

Это была реклама того, что надо везде ставить `var`.

2. Ошибка при рекурсии через функцию-свойство `window`.
Следующий код «умрет» в IE<9:

```
1 <script>
2 // рекурсия через функцию, явно записанную в
  window
3 window.recurse = function(times) {
4   if (times !== 0) recurse(times-1);
5 }
6
7 recurse(13);
8 </script>
```

Проблема здесь возникает из-за того, что функция напрямую присвоена в `window.recurse =` Ее не будет при обычном объявлении функции.

Этот пример выдаст ошибку только в настоящем IE8! Не IE9 в режиме эмуляции. Вообще, режим эмуляции позволяет отлавливать где-то 95% несовместимостей и проблем, а для

оставшихся 5% вам нужен будет настоящий IE8 в виртуальной машине.

Итого

В результате инициализации, к началу выполнения кода:

1. Функции, объявленные как `Function Declaration`, создаются полностью и готовы к использованию.
2. Переменные объявлены, но равны `undefined`. Присваивания выполняются позже, когда выполнение дойдет до них.



Задача: Window и переменная

Каков будет результат кода?

Важность: 5

```
1 | if ("a" in window) {  
2 |     var a = 1;  
3 | }  
4 | alert(a);
```

[Решение задачи "Window и переменная" »»](#)



Задача: Window и переменная 2

Каков будет результат (перед `a` нет `var`)?

Важность: 5

```
1 | if ("a" in window) {  
2 |     a = 1;  
3 | }  
4 | alert(a);
```

[Решение задачи "Window и переменная 2" »»](#)



Задача: Window и переменная 3

Каков будет результат (перед `a` нет `var`, а ниже есть)?

Важность: 5

```
1 | if ("a" in window) {
```

```
2   a = 1;
3   }
4   var a;
5
6   alert(a);
```

[Решение задачи "Window и переменная 3" »»](#)



Задача: Функция и переменная

Каков будет результат кода? Почему?

Важность: 3

```
1   var a = 5;
2
3   function a() { }
4
5   alert(a);
```

P.S. Это задание — учебное, на понимание процесса инициализации и выполнения. В реальной жизни мы, конечно же, не будем называть переменную и функцию одинаково.

[Решение задачи "Функция и переменная" »»](#)

Замыкания, функции изнутри

В этой главе мы продолжим рассматривать, как работают переменные, и, как следствие, познакомимся с замыканиями. От глобального объекта мы переходим к работе внутри функций.

Лексическое окружение

Все переменные внутри функции — это свойства специального внутреннего объекта `LexicalEnvironment`.

Мы будем называть этот объект «лексическое окружение» или просто «объект переменных».

При запуске функция создает объект `LexicalEnvironment`, записывает туда аргументы, функции и переменные. Процесс инициализации

выполняется в том же порядке, что и для глобального объекта, который, вообще говоря, является частным случаем лексического окружения.

В отличие от `window`, объект `LexicalEnvironment` является внутренним, он скрыт от прямого доступа.

Пример

Посмотрим пример, чтобы лучше понимать, как это работает:

```
1 function sayHi(name) {  
2   var phrase = "Привет, " + name;  
3   alert(phrase);  
4 }  
5  
6 sayHi('Вася');
```

При вызове функции:

1. До выполнения первой строчки её кода, на стадии инициализации, интерпретатор создает пустой объект `LexicalEnvironment` и заполняет его.

В данном случае туда попадает аргумент `name` и единственная переменная `phrase`:

```
1 function sayHi(name) {  
2   // LexicalEnvironment = { name: 'Вася', phrase:  
3   undefined }  
4   var phrase = "Привет, " + name;  
5   alert(phrase);  
6 }  
7 sayHi('Вася');
```

2. Функция выполняется.

Во время выполнения происходит присвоение локальной переменной `phrase`, то есть, другими словами, присвоение свойству `LexicalEnvironment.phrase` нового значения:

```
1 function sayHi(name) {  
2   // LexicalEnvironment = { name: 'Вася', phrase:
```

```

undefined }
3   var phrase = "Привет, " + name;
4
5   // LexicalEnvironment = { name: 'Вася', phrase:
   'Привет, Вася'}
6   alert(phrase);
7 }
8
9 sayHi( 'Вася' );

```

3. В конце выполнения функции объект с переменными обычно выбрасывается и память очищается.



Тонкости спецификации

Если почитать спецификацию ECMA-262, то мы увидим, что речь идёт о двух объектах: `VariableEnvironment` и `LexicalEnvironment`.

Но там же замечено, что в реализациях эти два объекта могут быть объединены. Так что мы избегаем лишних деталей и используем везде термин `LexicalEnvironment`, это достаточно точно позволяет описать происходящее.

Более формальное описание находится в спецификации ECMA-262, секции 10.2-10.5 и 13.

Доступ ко внешним переменным

Из функции мы можем обратиться не только к локальной переменной, но и к внешней:

```

1  var a = 5;
2
3  function f() {
4    alert(a); // 5
5  }

```

Интерпретатор, при доступе к переменной, сначала пытается найти переменную в текущем `LexicalEnvironment`, а затем, если её нет — ищет во внешнем объекте переменных. В данном случае им является

window.

Такой порядок поиска возможен благодаря тому, что ссылка на внешний объект переменных хранится в специальном внутреннем свойстве функции, которое называется `[[Scope]]`.

Рассмотрим, как оно создаётся и используется в коде выше:

1. Все начинается с момента создания функции. Функция создается не в вакууме, а в некотором лексическом окружении.

В случае выше функция создается в глобальном лексическом окружении `window`:

```
var a = 5  
function f() {  
  alert(a)  
}
```

LexicalEnvironment для этого кода
window = {a: ..., f: ...}

Для того, чтобы функция могла в будущем обратиться к внешним переменным, в момент создания она получает скрытое свойство `[[Scope]]`, которое ссылается на лексическое окружение, в котором она была создана:

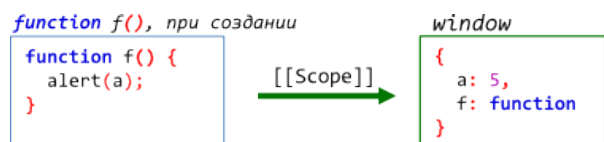
```
var a = 5  
function f() {  
  alert(a)  
}
```

`f.[[Scope]] = window`

Эта ссылка появляется одновременно с функцией и умирает вместе с ней. Программист не может как-либо получить или изменить её.

2. Позже, приходит время и функция запускается.

Интерпретатор вспоминает, что у неё есть свойство `f.[[Scope]]`:



...И использует его при создании объекта переменных для функции.

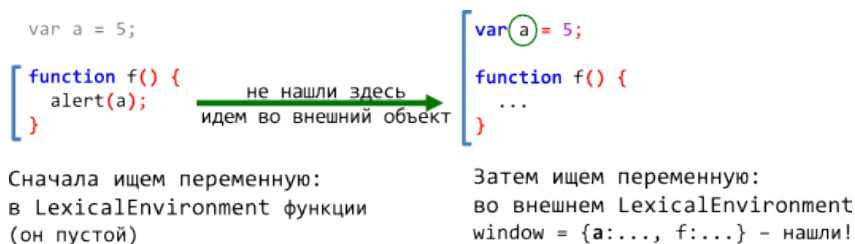
Новый объект `LexicalEnvironment` получает ссылку на «внешнее лексическое окружение» со значением из `[[Scope]]`. Эта ссылка используется для поиска переменных, которых нет в текущей функции.



Например, `alert(a)` сначала ищет в текущем объекте переменных: он пустой. А потом, как показано зеленой стрелкой на рисунке ниже — по ссылке, во внешнем окружении.



На уровне кода это выглядит как поиск во внешней области видимости, вне функции:



Если обобщить:

- Каждая функция при создании получает ссылку `[[Scope]]` на объект с переменными, в контексте которого была создана.
- При запуске функции создается новый объект с переменными. В него копируется ссылка на внешний объект из `[[Scope]]`.
- При поиске переменных он осуществляется сначала в текущем объекте переменных, а потом — по этой ссылке. Благодаря этому в функции доступны внешние переменные.



Задача: вопрос по `var`

Каков будет результат выполнения этого кода?

Важность: 5

```
01 var value = 0;
02
03 function f() {
04   if (1) {
05     value = true;
```

```
06     } else {  
07         var value = false;  
08     }  
09  
10     alert(value);  
11 }  
12  
13 f();
```

Изменится ли внешняя переменная `value` ?

P.S. Какими будут ответы, если из строки `var value = false` убрать `var`?

[Решение задачи "вопрос по var" »»](#)



Задача: `var window`

Каков будет результат выполнения этого кода?

Важность: 5

Почему?

```
01 function test() {  
02  
03     alert(window);  
04  
05     var window = 5;  
06  
07     alert(window);  
08 }  
09  
10 test();
```

[Решение задачи "var window" »»](#)



Задача: Вызов "на месте"

Каков будет результат выполнения кода? Почему?

Важность: 4

```
1 var a = 5  
2  
3 (function() {  
4     alert(a)  
5 })()
```

P.S. Подумайте хорошо! Здесь все ошибаются!

P.P.S. Внимание, здесь подводный камень! Ок, вы предупреждены.

Решение задачи "Вызов "на месте"" »»

Вложенные функции

Внутри функции можно объявлять не только локальные переменные, но и другие функции.

Как правило, это делают для вспомогательных операций, например в коде ниже для генерации сообщения используются `makeMessage` и `getHello`:

```
01 function sayHi(person) {
02
03     var message = makeMessage(person);
04     alert( message );
05
06     // ----- вспомогательные функции -----
07
08     function getHello(age) {
09         return age >= 18 ? 'Здравствуйте' : 'Привет';
10     }
11
12     function makeMessage(person) {
13         return getHello(person.age) + ', ' + person.name;
14     }
15 }
16
17 sayHi({
18     name: 'Петька',
19     age: 17
20 }); // привет, Петька
```

Вложенные функции могут быть объявлены и как `Function Declaration` и как `Function Expression`.

Вложенные функции обрабатываются в точности так же, как и глобальные. Единственная разница — они создаются в объекте переменных внешней функции, а не в `window`.

То есть, при запуске внешней функции `sayHi`, в её `LexicalEnvironment`

попадают локальные переменные и вложенные Function Declaration.

При этом Function Declaration сразу готовы к выполнению.

В примере выше при запуске sayHi(person) будет создан такой LexicalEnvironment:

```
1 | LexicalEnvironment = {  
2 |   person: переданный аргумент,  
3 |   message: undefined,  
4 |   getHello: function...,  
5 |   makeMessage: function...  
6 | }
```

Затем, во время выполнения sayHi, к вложенным функциям можно обращаться, они будут взяты из локального объекта переменных.

Вложенная функция имеет доступ к внешним переменным через [[Scope]].

1. При создании любая функция, в том числе и вложенная, получает свойство [[Scope]], указывающее на объект переменных, в котором она была создана.
2. При запуске она будет искать переменные сначала у себя, потом во внешнем объекте переменных, затем в более внешнем, и так далее.

Поэтому в примере выше из объявления функции makeMessage(person) можно убрать аргумент person.

Было:

```
1 | function sayHi(person) {  
2 |   ...  
3 |   function makeMessage(person) {  
4 |     return getHello(person.age) + ', ' + person.name;  
5 |   }  
6 | }
```

Станет:

```
1 | function sayHi(person) {  
2 |   ...  
3 |   function makeMessage() { // убрали аргумент
```

```

4      // переменная person будет взята из внешнего объекта
      переменных
5      return getHello(person.age) + ', ' + person.name;
6  }
7  }

```

Вложенную функцию можно вернуть.

Например, пусть sayHi не выдаёт alert тут же, а возвращает функцию, которая это делает:

```

01  function sayHi(person) {
02
03      return function() { // (*)
04          var message = makeMessage(person); // (**)
05          alert( message );
06      };
07
08      // ----- вспомогательные функции -----
09
10      function getHello(age) {
11          return age >= 18 ? 'Здравствуйте' : 'Привет';
12      }
13
14      function makeMessage() {
15          return getHello(person.age) + ', ' + person.name;
16      }
17  }
18
19  var sayHiPete = sayHi({ name: 'Петька', age: 17 });
20  var sayHiOther = sayHi({ name: 'Василий Иванович', age: 35
21  });
22  sayHiPete(); // эта функция может быть вызвана позже

```

В реальной жизни это нужно, чтобы вызвать получившуюся функцию позже, когда это будет нужно. Например, при нажатии на кнопку.

Возвращаемая функция (*) при запуске будет иметь полный доступ к аргументам внешней функции, а также к другим вложенным функциям makeMessage и getHello, так как при создании она получает ссылку [[Scope]], которая указывает на текущий LexicalEnvironment.

Переменные, которых нет в ней, например, person, будут взяты из него.

В частности, функция при вызове в строке будет взята

makeMessage

(**)

из внешнего объекта переменных.

Внешний `LexicalEnvironment`, в свою очередь, может ссылаться на ещё более внешний `LexicalEnvironment`, и так далее.

Замыканием функции называется сама эта функция, плюс вся цепочка `LexicalEnvironment`, которая при этом образуется.

Иногда говорят «переменная берётся из замыкания». Это означает — из внешнего объекта переменных.

Можно сказать и по-другому: «замыкание — это функция и все внешние переменные, которые ей доступны».

Управление памятью

JavaScript устроен так, что любой объект и, в частности, функция, существует до тех пор, пока на него есть ссылка, пока он как-то доступен для вызова, обращения. Более подробно об управлении памятью — далее, в статье [Управление памятью в JS и DOM](#).

Отсюда следует важное следствие при работе с замыканиями.

Объект переменных внешней функции существует в памяти до тех пор, пока существует хоть одна внутренняя функция, ссылающаяся на него через свойство `[[Scope]]`.

Посмотрим на примеры.

→ Обычно объект переменных удаляется по завершении работы функции. Даже если в нём есть объявление внутренней функции:

```
1 function f() {  
2   var value = 123;  
3   function g() { } // g видна только изнутри  
4 }  
5
```

```
6 | f();
```

В коде выше внутренняя функция объявлена, но она осталась внутри. После окончания работы `f()` она станет недоступной для вызовов, так что будет убрана из памяти вместе с остальными локальными переменными.

→ ...А вот в этом случае лексическое окружение, включая переменную `a`, будет сохранено:

```
1 | function f() {  
2 |   var a = Math.random();  
3 |  
4 |   function g() { }  
5 |  
6 |   return g;  
7 | }  
8 |  
9 | var g = f(); // функция g будет жить и сохранит ссылку  
   | на объект переменных
```

→ Если `f()` будет вызываться много раз, а полученные функции будут сохраняться, например, складываться в массив, то будут сохраняться и объекты `LexicalEnvironment` с соответствующими значениями `a`:

```
1 | function f() {  
2 |   var a = Math.random();  
3 |  
4 |   return function() { };  
5 | }  
6 |  
7 | // 3 функции,  
8 | // каждая ссылается на соответствующий объект  
   | LexicalEnvironment = {a: ...}  
9 | var arr = [f(), f(), f()];
```

Обратим внимание, что переменная `a` не используется в возвращаемой функции. Это означает, что браузерный оптимизатор может «де-факто» удалять её из памяти. Всё равно ведь никто не заметит.

→ В этом коде замыкание сначала сохраняется в памяти, а после удаления ссылки на `g` умирает:

```

01 function f() {
02     var a = Math.random();
03
04     function g() { }
05
06     return g;
07 }
08
09 var g = f(); // функция g жива
10 // а значит в памяти остается соответствующий объект
    переменных
11
12 g = null;    // ..а вот теперь память будет очищена

```

[[Scope]] для new Function

Есть одно исключение из общего правила присвоения [[Scope]].

При создании функции с использованием `new Function`, её свойство `[[Scope]]` ссылается не на текущий `LexicalEnvironment`, а на `window`.

Следующий пример демонстрирует как функция, созданная `new Function`, игнорирует внешнюю переменную `a` и выводит глобальную вместо нее.

Сначала обычное поведение:

```

01 var a = 1;
02 function getFunc() {
03     var a = 2;
04
05     var func = function() { alert(a); };
06
07     return func;
08 }
09
10 getFunc()(); // 2, из LexicalEnvironment функции getFunc

```

А теперь — для функции, созданной через `new Function`:

```

01 var a = 1;
02 function getFunc() {

```

```
03 |   var a = 2;
04 |
05 |   var func = new Function('', 'alert(a)');
06 |
07 |   return func;
08 | }
09 |
10 | getFunc(); // 1, из window
```

Итого

1. Все переменные и параметры функций являются свойствами объекта переменных `LexicalEnvironment`. Каждый запуск функции создает новый такой объект.
На верхнем уровне роль `LexicalEnvironment` играет «глобальный объект», в браузере это `window`.
2. При создании функция получает системное свойство `[[Scope]]`, которое ссылается на `LexicalEnvironment`, в котором она была создана (кроме `new Function`).
3. При запуске функции её `LexicalEnvironment` ссылается на внешний, сохраненный в `[[Scope]]`. Переменные сначала ищутся в своём объекте, потом — в объекте по ссылке и так далее, вплоть до `window`.

Разрабатывать без замыканий в JavaScript почти невозможно. Вы еще не раз встретитесь с ними в следующих главах учебника.

Хранение данных в замыканиях, модули

Замыкания можно использовать сотнями способов. Иногда люди сами не замечают, что использовали замыкания — настолько это просто и естественно.

В этой главе мы рассмотрим примеры использования замыканий для хранения данных и задачи на эту тему.

Данные для счётчика

В примере ниже makeCounter создает функцию, которая считает свои вызовы:

```
01 function makeCounter() {
02   var currentCount = 0;
03
04   return function() {
05     currentCount++;
06     return currentCount;
07   };
08 }
09
10 var counter = makeCounter();
11
12 // каждый вызов увеличивает счётчик
13 counter();
14 counter();
15 alert( counter() ); // 3
```

Хранение текущего числа вызовов осуществляется в переменной currentCount внешней функции.

При этом, так как каждый вызов makeCounter создает новый объект переменных, то все создаваемые функции-счётчики взаимно независимы.

```
1 var c1 = makeCounter();
2
3 var c2 = makeCounter();
4
5 alert( c1() ); // 1
6 alert( c2() ); // 1, счётчики независимы
```

Добавим счётчику аргумент, который, если передан, устанавливает значение:

```
01 function makeCounter() {
02   var currentCount = 0;
03
04   return function(newCount) {
05     if (newCount !== undefined) { // есть аргумент?
06       currentCount = +newCount; // сделаем его новым
07       значением счётчика
08     }
09     currentCount++;
10     return currentCount;
11   };
12 }
```

```

07     // вернём текущее значение, счётчик всегда возвращает
    его (это удобно)
08     return currentCount;
09 }
10
11     currentCount++;
12     return currentCount;
13 };
14 }
15
16 var counter = makeCounter();
17
18 alert( counter() ); // 1
19 alert( counter(3) ); // 3
20 alert( counter() ); // 4

```

Здесь для проверки первого аргумента использовано сравнение с `undefined`, так что вызовы `counter(undefined)` и `counter()` сработают идентично, как будто аргументов нет.

Объект счётчика

Можно пойти дальше и вернуть полноценный объект с функциями управления счётчиком:

- `getNext()` — получить следующее значение, то, что раньше делал вызов `counter()`.
- `set(value)` — поставить значение.
- `reset()` — обнулить счётчик.

```

01 function makeCounter() {
02     var currentCount = 0;
03
04     return {
05         getNext: function() {
06             return ++currentCount;
07         },
08
09         set: function(value) {
10             currentCount = value;
11         },
12
13         reset: function() {

```

```

14     currentCount = 0;
15 }
16 };
17 }
18
19 var counter = makeCounter();
20
21 alert( counter.getNext() ); // 1
22 alert( counter.getNext() ); // 2
23
24 counter.reset();
25 alert( counter.getNext() ); // 1

```

...Теперь `counter` — объект с методами, которые при работе используют `currentCount`. Снаружи никак иначе, кроме как через эти методы, к `currentCount` получить доступ нельзя, так как это локальная переменная.

Объект счётчика + функция

К сожалению, пропал короткий красивый вызов `counter()`, вместо него теперь `counter.getNext()`. Но он ведь был таким коротким и удобным... Так что давайте вернём его:

```

01 function makeCounter() {
02     var currentCount = 0;
03
04     // возвращаемся к функции
05     function counter() {
06         return ++currentCount;
07     }
08
09     // ...и добавляем ей методы!
10     counter.set = function(value) {
11         currentCount = value;
12     };
13
14     counter.reset = function() {
15         currentCount = 0;
16     };
17
18     return counter;
19 }
20
21 var counter = makeCounter();

```

```
22  
23 alert( counter() ); // 1  
24 alert( counter() ); // 2  
25  
26 counter.reset();  
27 alert( counter() ); // 1
```

Красиво, не правда ли? Тот факт, что объект — функция, вовсе не мешает добавить к нему сколько угодно методов.

Приём проектирования «Модуль»

Приём программирования «модуль» имеет громадное количество вариаций.

Его цель — объявить функции так, чтобы ненужные подробности их реализаций были скрыты. В том числе, временные переменные, константы, вспомогательные мини-функции и т.п.

Оформление кода в модуль предусматривает следующие шаги:

1. Создаётся функция-обёртка, которая выполняется «на месте»:

```
(function() {  
    ...  
})();
```

2. Внутри этой функции пишутся локальные переменные и функции, которые пользователю модуля не нужны, но нужны самому модулю:

```
1 (function() {  
2     var count = 0;  
3  
4     function helper() { ... }  
5 })();
```

Они будут доступны только изнутри.

3. Те функции, которые нужны пользователю, «экспортируются» во внешнюю область видимости.

Если функция одна — это можно сделать явным возвратом `return`:


```

1  var func = (function() {
2      var count = 0;
3      function helper() { ... }
4
5      return function() { ... }
6  })();

```

...Если функций много — можно присвоить их напрямую в window:

```

1  (function() {
2
3      function helper() { ... }
4
5      window.createMenu = function() { ... };
6      window.createDialog = function() { ... };
7
8  })();

```

Или, что ещё лучше, вернуть объект с ними:

```

01  var MyLibrary = (function() {
02
03      function helper() { ... }
04
05      return {
06          createMenu: function() { ... },
07          createDialog: function() { ... }
08      };
09
10  })();
11
12  // использование
13  MyLibrary.createMenu();

```

Все функции модуля будут через замыкание иметь доступ к другим переменным и внутренним функциям. Но снаружи программист, использующий модуль, может обращаться напрямую только к тем, 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

Задачи на понимание замыканий



Задача: Сумма через замыкание

Напишите функцию `sum`, которая работает так:

Важность: 4

`sum(a)(b) = a+b.`

Да, именно так, через двойные скобки (это не опечатка).

Например:

`sum(1)(2) = 3`

`sum(5)(-1) = 4`

[Решение задачи "Сумма через замыкание" »»](#)



Задача: Фильтрация через функцию

Важность: 5

1. Создайте функцию `filter(arr, func)`, которая получает массив `arr` и возвращает новый, в который входят только те элементы `arr`, для которых `func` возвращает `true`.
2. Создайте набор «готовых фильтров»: `inBetween(a,b)` — «между `a,b`», `inArray([...])` — «в массиве `[...]`».

Использование должно быть таким:

- `filter(arr, inBetween(3,6))` — выберет только числа от 3 до 6,
- `filter(arr, inArray([1,2,3]))` — выберет только элементы, совпадающие с одним из значений массива.

Пример, как это должно работать:

```
1  /* .. ваш код для filter, inBetween, inArray */
2  var arr = [1, 2, 3, 4, 5, 6, 7];
3
4  alert( filter(arr, function(a) { return a % 2 == 0
5    }) ); // 2,4,6
6
6  alert( filter(arr, inBetween(3,6)) ); // 3,4,5,6
```

7

```
8 alert( filter(arr, inArray([1,2,10])) ); // 1,2
```

[Решение задачи "Фильтрация через функцию" »»](#)



Задача: Функция - строковый буфер

В некоторых языках программирования существует объект «строковый буфер», который аккумулирует внутри себя значения. Его функционал состоит из двух возможностей:

Важность: 5

1. Добавить значение в буфер.
2. Получить текущее содержимое.

Задача — реализовать строковый буфер на функциях в JavaScript, со следующим синтаксисом:

- Создание объекта: `var buffer = makeBuffer();`.
- Вызов `makeBuffer` должен возвращать такую функцию `buffer`, которая при вызове `buffer(value)` добавляет значение в некоторое внутреннее хранилище, а при вызове без аргументов `buffer()` — возвращает его.

Вот пример работы:

```
01 function makeBuffer() { /* ваш код */ }
02
03 var buffer = makeBuffer();
04
05 // добавить значения к буферу
06 buffer( 'Замыкания' );
07 buffer( ' Использовать' );
08 buffer( ' Нужно!' );
09
10 // получить текущее значение
11 alert( buffer() ); // Замыкания Использовать Нужно!
```

Буфер должен преобразовывать все данные к строковому типу:

```
1 var buffer = makeBuffer();
2 buffer(0); buffer(1); buffer(0);
3
4 alert( buffer() ); // '010'
```

Решение не должно использовать глобальные переменные.

[Решение задачи "Функция - строковый буфер" »»](#)



Задача: Строковый буфер с методами

Добавьте буфферу из решения задачи [Функция - строковый буфер](#) метод `buffer.clear()`, который будет очищать текущее содержимое буфера:

Важность: 5

```
01 function makeBuffer() {
02     ...ваш код...
03 }
04
05 var buffer = makeBuffer();
06
07 buffer("Тест");
08 buffer(" тебя не съест ");
09 alert( buffer() ); // Тест тебя не съест
10
11 buffer.clear();
12
13 alert( buffer() ); // ""
```

[Решение задачи "Строковый буфер с методами" »»](#)



Задача: Будет ли доступ через замыкание?

Важность: 5

- Будет ли работать доступ к переменной `name` через замыкание в примере ниже?
- Удалится ли переменная `name` из памяти при выполнении `delete donkey.sayHi`? Если нет — можно ли к `name` как-то обратиться после удаления `donkey.sayHi`?
- А если присвоить `donkey.sayHi = donkey.yell = null` — останется ли `name` в памяти?

```

01 var makeDonkey = function() {
02     var name = "Ослик Иа";
03
04     return {
05         sayHi: function() {
06             alert(name);
07         },
08         yell: function() {
09             alert('И-а, и-а!');
10         }
11     };
12 }
13
14 var donkey = makeDonkey();
15 donkey.sayHi();

```

Решение задачи "Будет ли доступ через замыкание?" »»



Задача: Армия функций

Следующий код создает массив функций-стрелков shooters. По замыслу, каждый стрелок должен выводить свой номер:

Важность: 5

```

01 function makeArmy() {
02
03     var shooters = [];
04
05     for(var i=0; i<10; i++) {
06         var shooter = function() { // функция-стрелок
07             alert(i); // выводит свой номер
08         };
09         shooters.push(shooter);
10     }
11
12     return shooters;
13 }
14
15 var army = makeArmy();
16
17 army[0](); // стрелок выводит 10, а должен 0
18 army[5](); // стрелок выводит 10...
19 // .. все стрелки выводят 10 вместо 0,1,2...9

```

Почему все стрелки выводят одно и то же? Поправьте код, чтобы

стрелки работали как задумано. Предложите несколько вариантов исправления.

[Решение задачи "Армия функций" »»](#)

Статические переменные

Статическая переменная функции — это такая, которая сохраняет значение между вызовами.

Такие переменные есть во многих языках. В JavaScript они не реализованы синтаксически, но можно организовать аналог.

Использование замыкания

В предыдущей главе мы видели, как реализовать статическую переменную, используя замыкание.

В примере ниже количество `count` вызовов функции `sayHi` сохраняется в обёртке:

```
01 var sayHi = (function() {  
02  
03     var count = 0; // статическая переменная  
04  
05     return function() {  
06         count++;  
07  
08         alert("Привет " + count);  
09     };  
10  
11 })();  
12  
13 sayHi(); // Привет 1  
14 sayHi(); // Привет 2
```

Это достаточно хороший способ, но, в качестве альтернативы, рассмотрим ещё один.

Запись в функцию

Благодаря тому, что функция — это объект, можно добавить статические свойства прямо к ней.

Перепишем пример, используя запись в функцию:

```
01 function sayHi() {  
02     sayHi.count++;  
03  
04     alert("Привет " + sayHi.count);  
05 }  
06  
07 sayHi.count = 0; // начальное значение  
08  
09 sayHi(); // Привет 1  
10 sayHi(); // Привет 2
```

Как видно, пример работает также, но внутри всё по-другому.

Статическая переменная, записанная как свойство функции — общедоступна. К ней имеет доступ любой, у кого есть объект функции.

Этим она отличается от привязки через замыкание.



Глобальная переменная?

В примере выше можно было бы вынести количество вызовов в глобальную переменную `sayHiCount`. Но это лишь потому, что пример прост.

Глобальные переменные, вообще, следует использовать пореже. Статическую переменную естественно связывать напрямую с функцией, что и решается замыканием, либо записью в свойство.



Задача: Создать счётчик со статической переменной

Создайте счётчик, который вместо замыкания использует статическую переменную для хранения `currentCount`.

Важность: 3

Код счётчика, работающий через замыкание (для переделки):

```
1 function makeCounter() {  
2     // currentCount можно считать "статической  
   переменной" счётчика  
3     var currentCount = 0;  
4  
5     return function() {  
6         currentCount++;  
7         return currentCount;  
8     };  
9 }
```

Важно: счётчики должны быть независимыми:

```
1 var c1 = makeCounter();  
2 var c2 = makeCounter();  
3  
4 alert( c1() ); // 1  
5 alert( c2() ); // 1  
6 alert( c1() ); // 2
```

Решение задачи "Создать счётчик со статической переменной"
»»

Конструкция "with"

Конструкция `with` позволяет использовать для области видимости произвольный объект. В современном JavaScript от этой конструкции отказались, но её ещё можно найти в старом коде.

Синтаксис:

```
with(obj) {  
    ... КОД ...  
}
```

Любое обращение к переменной внутри `with` сначала ищет её среди свойств `obj`, а только потом — вне `with`.

Пример

В примере ниже переменная будет взята не из глобальной области, а из obj:

```
1  var a = 5;
2
3  var obj = { a : 10 };
4
5  with(obj) {
6    alert(a); // 10, из obj
7  }
```

Попробуем получить переменную, которой в obj нет:

```
1  var b = 1;
2
3  var obj = { a : 10 };
4
5  with(obj) {
6    alert(b); // 1, из window
7  }
```

Здесь интерпретатор сначала проверяет наличие obj.b, не находит и идет вне with.

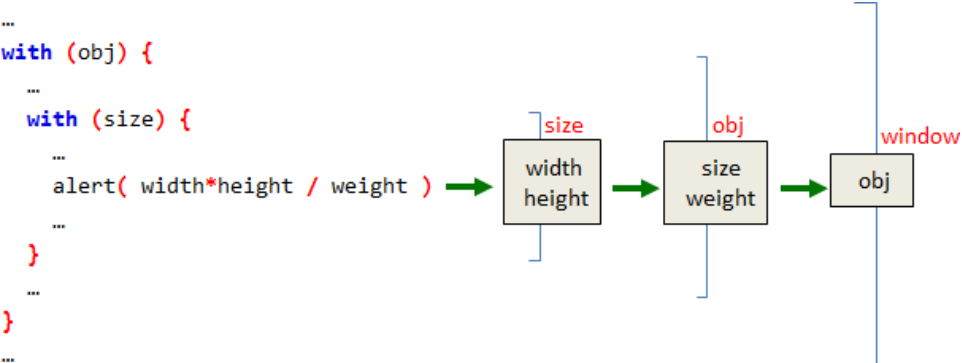
Особенно забавно выглядит применение вложенных with:

```
01  var obj = {
02    weight: 10,
03    size: {
04      width: 5,
05      height: 7
06    }
07  };
08
09  with(obj) {
10    with(size) { // size будет взят из obj
11      alert( width*height / weight ); // width,height из
12      size, weight из obj
13    }
14  }
```

Свойства из разных объектов используются как обычные переменные...

Магия! Порядок поиска переменных в выделенном коде:

size => obj => window



Изменения переменной

При использовании `with`, как и во вложенных функциях — переменная изменяется в той области, где была найдена.

Например:

```
1 var obj = { a : 10 }  
2  
3 with(obj) {  
4   a = 20;  
5 }  
6 alert(obj.a); // 20, переменная была изменена в объекте
```

Почему отказались от with?

Есть несколько причин.

1. В современном стандарте JavaScript отказались от `with`, потому что **конструкция `with` подвержена ошибкам и непрозрачна.**

Проблемы возникают в том случае, когда в `with(obj)` присваивается переменная, которая по замыслу должна быть в свойствах `obj`, но ее там нет.

Например:

```
1 var obj = { weight: 10 };  
2  
3 with(obj) {  
4   weight = 20; // (1)  
5   size = 35;   // (2)  
6 }  
7  
8 alert(obj.size);
```

```
9 | alert(window.size);
```

В строке (2) присваивается свойство, отсутствующее в `obj`. В результате интерпретатор, не найдя его, создает новую глобальную переменную `window.size`.

Такие ошибки редки, но очень сложны в отладке, особенно если `size` изменилась не в `window`, а где-нибудь во внешнем `LexicalEnvironment`.

- Еще одна причина — **алгоритмы сжатия JavaScript не любят `with`**. Перед выкладкой на сервер JavaScript сжимают. Для этого есть много инструментов, например [Closure Compiler](#) и [UglifyJS](#). Если вкратце — они либо сжимают код с `with` с ошибками, либо оставляют его частично несжатым.
- Ну и, наконец, **производительность — усложнение поиска переменной из-за `with` влечет дополнительные накладные расходы**. Современные движки применяют много внутренних оптимизаций, ряд которых не могут быть применены к коду, в котором есть `with`.

Вот, к примеру, запустите этот код в современном браузере.

Производительность функции `fast` существенно отличается `slow` с пустым(!) `with`. И дело тут именно в `with`, т.к. наличие этой конструкции препятствует оптимизации.

```
01 | var i = 0;
02 |
03 | function fast() {
04 |     i++;
05 | }
06 |
07 | function slow() {
08 |     with(i) {}
09 |     i++;
10 | }
11 |
12 |
13 | var time = new Date();
14 | while(i < 1000000) fast();
15 | alert(new Date - time);
```

```
16  
17 var time = new Date();  
18 i=0;  
19 while(i < 1000000) slow();  
20 alert(new Date - time);
```

Замена with

Вместо with рекомендуется использовать временную переменную, например:

```
01 /* вместо  
02 with(elem.style) {  
03     top = '10px';  
04     left = '20px';  
05 }  
06 */  
07  
08 var s = elem.style;  
09  
10 s.top = '10px';  
11 s.left = '0';
```

Это не так элегантно, но убирает лишний уровень вложенности и абсолютно точно понятно, что будет происходить и куда присвоятся свойства.

Итого

- Конструкция with(obj) { ... } использует obj как дополнительную область видимости. Все переменные, к которым идет обращение внутри блока, сначала ищутся в obj.
- Конструкция with устарела и не рекомендуется по ряду причин. Избегайте её.



Задача: With + функция

Какая из функций будет вызвана?

Важность: 5

```
1 function f() { alert(1) }  
2
```

```
3 var obj = {  
4   f: function() { alert(2) }  
5 };  
6  
7 with(obj) {  
8   f();  
9 }
```

[Решение задачи "With + функция" »»](#)



Задача: With + переменные

Что выведет этот код?

Важность: 5

```
1 var a = 1;  
2  
3 var obj = { b: 2 };  
4  
5 with(obj) {  
6   var b;  
7   alert( a + b );  
8 }
```

[Решение задачи "With + переменные" »»](#)

Решения задач



Решение задачи: Проверка на NFE

Первый код выведет `function ...`, второй — ошибку во всех браузерах, кроме IE8-.

```
1 // обычное объявление функции (Function Declaration)  
2 function g() { return 1; };  
3  
4 alert(g); // функция
```

Во втором коде скобки есть, значит функция внутри является не `Function Declaration`, а частью выражения, то есть `Named Function Expression`. Его имя видно только внутри,

снаружи переменная `g` не определена.

```
1 // Named Function Expression!  
2 (function g() { return 1; });  
3  
4 alert(g); // Ошибка!
```

Все браузеры, кроме IE8-, поддерживают это ограничение видимости и выведут ошибку, 'undefined variable'.



Решение задачи: Перезапись NFE по имени

При работе без `use strict` ничего особенного не произойдет, функция будет работать как обычно:

```
1 var f = function sayHi(n) {  
2   alert(n);  
3   sayHi = 0;  
4 };  
5  
6 f(1); // 1  
7 f(2); // 2
```

Дело в том, что переменная `f` хранит ссылку на функцию.

Изнутри функции создается переменная `sayHi`, которая тоже ссылается на функцию. Если мы перезапишем её, ничего страшного не случится.

В современном стандарте указано, что такую переменную изменить нельзя.

Поэтому в строгом режиме будет ошибка, если конечно браузер корректно реализует эту часть стандарта:

```
1 var f = function sayHi() {  
2   "use strict";  
3  
4   sayHi = 0; // ошибка в современных браузерах  
5   alert("При корректной работе интерпретатора мы не  
6     увидим этот alert");  
7 }  
8
```

```
8 | f();
```



Решение задачи: Window и переменная

Ответ: 1.

```
1 | if ("a" in window) {  
2 |     var a = 1;  
3 | }  
4 | alert(a);
```

Посмотрим, почему.

На стадии подготовки к выполнению, из `var a` создается `window.a`:

```
1 | // window = {a:undefined}  
2 |  
3 | if ("a" in window) { // в if видно что window.a уже  
   | есть  
4 |     var a = 1; // поэтому эта строка сработает  
5 | }  
6 | alert(a);
```

В результате `a` становится 1.



Решение задачи: Window и переменная 2

Ответ: **ошибка**.

Переменной `a` нет, так что условие `"a" in window` не выполнится. В результате на последней строчке - обращение к неопределенной переменной.

```
1 | if ("a" in window) {  
2 |     a = 1;  
3 | }  
4 | alert(a); // <-- error!
```



Решение задачи: Window и переменная 3

Ответ: 1.

Переменная `a` создается до начала выполнения кода, так что условие `"a" in window` выполнится и сработает `a = 1`.

```
1  if ("a" in window) {  
2      a = 1;  
3  }  
4  var a;  
5  
6  alert(a); // 1
```



Решение задачи: Функция и переменная

Ответ: 5.

```
1  var a = 5;  
2  
3  function a() { }  
4  
5  alert(a);
```

Чтобы понять, почему — разберём внимательно как работает этот код.

1. До начала выполнения создаётся переменная `a` и функция `a`. Стандарт написан так, что функция создаётся первой и переменная ее не перезаписывает. То есть, функция имеет приоритет. Но в данном случае это совершенно неважно, потому что...
2. ...После инициализации, когда код начинает выполняться — срабатывает присваивание `a = 5`, перезаписывая `a`, и уже не важно, что там лежало.
3. Объявление `Function Declaration` на стадии выполнения игнорируется (уже обработано).
4. В результате `alert(a)` выводит 5.



Решение задачи: вопрос по var

Результатом будет true, т.к. `var` обрабатывается и переменная будет создана до выполнения кода.

Соответственно, присвоение `value=true` сработает на локальной переменной, и `alert` выведет `true`.

Внешняя переменная не изменится.

P.S. Если `var` нет, то в функции переменная не будет найдена. Интерпретатор обратится за ней в `window` и изменит её там.

Так что без `var` результат будет также true, но внешняя переменная изменится.



Решение задачи: `var window`

Результатом будет `undefined`, затем 5.

```
01 function test() {  
02  
03     alert(window);  
04  
05     var window = 5;  
06  
07     alert(window);  
08 }  
09  
10 test();
```

Директива `var` обрабатывается до начала выполнения кода функции. Будет создана локальная переменная, т.е. свойство `LexicalEnvironment`:

```
LexicalEnvironment = {  
    window: undefined  
}
```

Когда выполнение кода начнется и сработает `alert`, он выведет локальную переменную.

Затем сработает присваивание, и второй `alert` выведет уже 5.





Решение задачи: Вызов "на месте"

Результат - **ошибка**. Попробуйте:

```
1 | var a = 5
2 |
3 | (function() {
4 |     alert(a)
5 | })()
```

Дело в том, что после `var a = 5` нет точки с запятой.

JavaScript воспринимает этот код как если бы перевода строки не было:

```
1 | var a = 5(function() {
2 |     alert(a)
3 | })()
```

То есть, он пытается вызвать *функцию* 5, что и приводит к ошибке.

Если точку с запятой поставить, все будет хорошо:

```
1 | var a = 5;
2 |
3 | (function() {
4 |     alert(a)
5 | })()
```

Это один из наиболее частых и опасных подводных камней, приводящих к ошибкам тех, кто *не* ставит точки с запятой.



Решение задачи: Сумма через замыкание

Чтобы вторые скобки в вызове работали - первые должны возвращать функцию.

Эта функция должна знать про `a` и уметь прибавлять `a` к `b`. Вот так:

```
01 | function sum(a) {
02 |
```

```

03     return function(b) {
04         return a + b; // ВОЗЬМЕТ a ИЗ ВНЕШНЕГО
    LexicalEnvironment
05     };
06
07 }
08
09 alert( sum(1)(2) );
10 alert( sum(5)(-1) );

```



Решение задачи: Фильтрация через функцию

Функция фильтрации

```

01 function filter(arr, func) {
02     var result = [];
03
04     for(var i=0; i<arr.length; i++) {
05         var val = arr[i];
06         if (func(val)) {
07             result.push(val);
08         }
09     }
10
11     return result;
12 }
13
14 var arr = [1, 2, 3, 4, 5, 6, 7];
15
16 alert( filter(arr, function(a) { return a % 2 == 0;
    }) ); // 2, 4, 6

```

Фильтр inBetween

```

01 function filter(arr, func) {
02     var result = [];
03
04     for(var i=0; i<arr.length; i++) {
05         var val = arr[i];
06         if (func(val)) {
07             result.push(val);
08         }
09     }
10

```

```

11     return result;
12 }
13
14 function inBetween(a, b) {
15     return function(x) {
16         return x >=a && x <= b;
17     };
18 }
19
20 var arr = [1, 2, 3, 4, 5, 6, 7];
21 alert( filter(arr, inBetween(3,6)) ); // 3,4,5,6

```

Фильтр inArray

```

01 function filter(arr, func) {
02     var result = [];
03
04     for(var i=0; i<arr.length; i++) {
05         var val = arr[i];
06         if (func(val)) {
07             result.push(val);
08         }
09     }
10
11     return result;
12 }
13
14 function inArray(arr) {
15     return function(x) {
16         return arr.indexOf(x) != -1;
17     };
18 }
19
20 var arr = [1, 2, 3, 4, 5, 6, 7];
21 alert( filter(arr, inArray([1,2,10])) ); // 1,2

```



Решение задачи: Функция - строковый буфер

Текущее значение текста удобно хранить в замыкании, в локальной переменной `makeBuffer`:

```

01 function makeBuffer() {
02     var text = '';
03

```

```

04  return function(piece) {
05      if (piece === undefined) { // вызов без
    аргументов
06          return text;
07      }
08      text += piece;
09  };
10  };
11
12  var buffer = makeBuffer();
13
14  // добавить значения к буферу
15  buffer('Замыкания');
16  buffer(' Использовать');
17  buffer(' Нужно!');
18  alert( buffer() ); // 'Замыкания Использовать
    Нужно!'
19
20  var buffer2 = makeBuffer();
21  buffer2(0); buffer2(1); buffer2(0);
22
23  alert( buffer2() ); // '010'

```

Начальное значение `text = ''` — пустая строка. Поэтому операция `text += piece` прибавляет `piece` к строке, автоматически преобразуя его к строковому типу, как и требовалось в условии.

Решение задачи: Строковый буфер с методами

```

01  function makeBuffer() {
02      var text = '';
03
04      function buffer(piece) {
05          if (piece === undefined) { // вызов без
    аргументов
06              return text;
07          }
08          text += piece;
09      };
10
11      buffer.clear = function() {
12          text = "";
13      }

```

```

14
15     return buffer;
16 };
17
18 var buffer = makeBuffer();
19
20 buffer("Тест");
21 buffer(" тебя не съест ");
22 alert( buffer() ); // Тест тебя не съест
23
24 buffer.clear();
25
26 alert( buffer() ); // ""

```

Решение задачи: Будет ли доступ через замыкание?

1. Да, будет работать, благодаря ссылке `[[Scope]]` на внешний объект переменных, которая будет присвоена функциям `sayHi` и `yell` при создании объекта.
2. Нет, `name` не удалится из памяти, поскольку несмотря на то, что `sayHi` больше нет, есть ещё функция `yell`, которая также ссылается на внешний объект переменных. Этот объект хранится целиком, вместе со всеми свойствами.

При этом, так как функция `sayHi` удалена из объекта и ссылок на нее нет, то больше к переменной `name` обращаться некому. Получилось, что она «застряла» в памяти, хотя, по сути, никому не нужна.

3. Если и `sayHi` и `yell` удалить, тогда, так как больше внутренних функций не останется, удалится и объект переменных вместе с `name`.

Решение задачи: Армия функций

Что происходит в этом коде

Функция `makeArmy` делает следующее:

1. Создает пустой массив `shooter`:

```
var shooters = [];
```

2. В цикле заполняет массив элементами через `shooter.push`. При этом каждый элемент массива — это функция, так что в итоге после цикла массив будет таким:

```
01 shooters = [  
02   function () { alert(i); },  
03   function () { alert(i); },  
04   function () { alert(i); },  
05   function () { alert(i); },  
06   function () { alert(i); },  
07   function () { alert(i); },  
08   function () { alert(i); },  
09   function () { alert(i); },  
10   function () { alert(i); },  
11   function () { alert(i); }  
12 ];
```

Этот массив возвращается из функции.

3. Вызов `army[5]()` — это получение элемента массива (им будет функция) на позиции 5, и тут же — её запуск.

Почему ошибка

Вначале разберемся, почему все стрелки выводят одно и то же значение.

В функциях-стрелках `shooter` отсутствует переменная `i`. Когда такая функция вызывается, то `i` она берет из внешнего `LexicalEnvironment...`

Каким будет значение `i`?

К моменту вызова `army[0]()`, функция `makeArmy` уже закончила работу. Цикл завершился, последнее значение было `i=10`.

В результате все функции `shooter` получают одно и то же, последнее, значение `i=10`.

Попробуйте исправить проблему самостоятельно.

Исправление (3 варианта)

Есть несколько способов исправить ситуацию.

1. Первый способ исправить код - это привязать значение непосредственно к функции-стрелку:

```
01 function makeArmy() {  
02  
03     var shooters = [];  
04  
05     for(var i=0; i<10; i++) {  
06  
07         var shooter = function me() {  
08             alert( me.i );  
09         };  
10         shooter.i = i;  
11  
12         shooters.push(shooter);  
13     }  
14  
15     return shooters;  
16 }  
17  
18 var army = makeArmy();  
19  
20 army[0](); // 0  
21 army[1](); // 1
```

В этом случае каждая функция хранит в себе свой собственный номер.

Кстати, обратите внимание на использование Named Function Expression, вот в этом участке:

```
1 ...  
2 var shooter = function me() {  
3     alert( me.i );  
4 };  
5 ...
```

Если убрать имя `me` и оставить обращение через `shooter`, то работать не будет:


```

1  for(var i=0; i<10; i++) {
2      var shooter = function() {
3          alert(shooter.i); // вывести свой номер (не
                             // работает!)
4          // потому что откуда функция возьмёт
                             // переменную shooter?
5          // ..правильно, из внешнего объекта, а там
                             // она одна на всех
6      };
7      shooter.i = i;
8      shooters.push(shooter);
9  }

```

Вызов `alert(shooter.i)` при вызове будет искать переменную `shooter`, а эта переменная меняет значение по ходу цикла, и к моменту вызову она равна последней функции, созданной в цикле.

Если использовать Named Function Expression, то имя жёстко привязывается к конкретной функции, и поэтому в коде выше `me.i` возвращает правильный `i`.

2. Другое, более продвинутое решение — использовать дополнительную функцию для того, чтобы «поймать» текущее значение `i`:

```

01  function makeArmy() {
02      var shooters = [];
03      for(var i=0; i<10; i++) {
04          var shooter = (function(x) {
05              return function() {
06                  alert( x );
07              };
08          })(i);
09      shooters.push(shooter);
10  }
11  return shooters;

```

```
19 }
20
21 var army = makeArmy();
22
23 army[0](); // 0
24 army[1](); // 1
```

Посмотрим выделенный фрагмент более внимательно, чтобы понять, что происходит:

```
1 var shooter = (function(x) {
2   return function() {
3     alert( x );
4   };
5 })(i);
```

Функция `shooter` создана как результат вызова промежуточного функционального выражения `function(x)`, которое объявляется — и тут же выполняется, получая `x = i`. Так как `function(x)` тут же завершается, то значение `x` больше не меняется. Оно и будет использовано в возвращаемой функции-стрелке.

Для красоты можно изменить название переменной `x` на `i`, суть происходящего при этом не изменится:

```
1 var shooter = (function(i) {
2   return function() {
3     alert( i );
4   };
5 })(i);
```

Кстати, обратите внимание — скобки вокруг `function(i)` не нужны, можно и так:

```
1 var shooter = function(i) { // без скобок
2   return function() {
3     alert( i );
4   };
5 }(i);
```

Скобки добавлены в код для лучшей читаемости, чтобы человек, который просматривает его, не подумал, что `var shooter = function`, а понял что это вызов «на месте», и присваивается его результат.

3. Еще один забавный способ - обернуть весь цикл во временную функцию:

```
01 function makeArmy() {  
02  
03     var shooters = [];  
04  
05     for(var i=0; i<10; i++) (function(i) {  
06  
07         var shooter = function() {  
08             alert( i );  
09         };  
10  
11         shooters.push(shooter);  
12  
13     })(i);  
14  
15     return shooters;  
16 }  
17  
18 var army = makeArmy();  
19  
20 army[0](); // 0  
21 army[1](); // 1
```

Вызов `(function(i) { ... })` обернут в скобки, чтобы интерпретатор понял, что это `Function Expression`.

Плюс этого способа - в большей читаемости. Фактически, мы не меняем создание `shooter`, а просто обертываем итерацию в функцию.



Решение задачи: Создать счётчик со статической переменной

Переписанный счётчик:

```
01 function makeCounter() {
```

```

02  return function f() {
03      if (!f.currentCount) {
04          f.currentCount = 0;
05      }
06
07      return ++f.currentCount;
08  };
09  }
10
11  var c1 = makeCounter();
12  var c2 = makeCounter();
13
14  alert( c1() ); // 1
15  alert( c2() ); // 1
16  alert( c1() ); // 2

```

Побочный эффект — текущее значение счётчика теперь доступно снаружи через свойство функции:

```

1  var counter = makeCounter();
2
3  counter();
4  alert( counter.currentCount ); // 1

```



Решение задачи: With + функция

Вторая (2), т.к. при обращении к любой переменной внутри with — она ищется прежде всего в объекте.

Соответственно, будет выведено 2:

```

1  function f() { alert(1) }
2
3  var obj = {
4      f: function() { alert(2) }
5  };
6
7  with(obj) {
8      f();
9  }

```



Решение задачи: With + переменные

Выведет 3.

Конструкция `with` не создаёт области видимости, её создают только функции. Поэтому объявление `var b` внутри конструкции работает также, как если бы оно было вне её.

Код в задаче эквивалентен такому:

```
1 var a = 1;
2 var b;
3
4 var obj = { b: 2 }
5
6 with(obj) {
7     alert( a + b );
8 }
```