**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](http://learn.javascript.ru/closures" \l "leksicheskoe-okruzhenie)**

Все переменные внутри функции – это свойства специального внутреннего объекта LexicalEnvironment, который создаётся при её запуске.

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](http://learn.javascript.ru/closures" \l "primer)**

Посмотрим пример, чтобы лучше понимать, как это работает:

function sayHi(name) {

var phrase = "Привет, " + name;

alert( phrase );

}

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:

function sayHi(name) {

// LexicalEnvironment = { name: 'Вася', phrase: undefined }

var phrase = "Привет, " + name;

alert( phrase );

}

sayHi('Вася');

1. Функция выполняется.

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

function sayHi(name) {

// LexicalEnvironment = { name: 'Вася', phrase: undefined }

var phrase = "Привет, " + name;

// LexicalEnvironment = { name: 'Вася', phrase: 'Привет, Вася'}

alert( phrase );

}

sayHi('Вася');

1. В конце выполнения функции объект с переменными обычно выбрасывается и память очищается. В примерах выше так и происходит. Через некоторое время мы рассмотрим более сложные ситуации, при которых объект с переменными сохраняется и после завершения функции.

**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](http://learn.javascript.ru/closures" \l "dostup-ko-vneshnim-peremennym)**

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

var userName = "Вася";

function sayHi() {

alert( userName ); // "Вася"

}

**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]]. Это свойство закрыто от прямого доступа, но знание о нём очень важно для понимания того, как работает JavaScript.

**При создании функция получает скрытое свойство [[Scope]], которое ссылается на лексическое окружение, в котором она была создана.**

В примере выше таким окружением является window, так что создаётся свойство:

sayHi.[[Scope]] = window

Это свойство никогда не меняется. Оно всюду следует за функцией, привязывая её, таким образом, к месту своего рождения.

При запуске функции её объект переменных LexicalEnvironment получает ссылку на «внешнее лексическое окружение» со значением из [[Scope]].

Если переменная не найдена в функции – она будет искаться снаружи.

Именно благодаря этой механике в примере выше alert(userName) выводит внешнюю переменную. 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 создаётся новый объект с переменными LexicalEnvironment. Он получает ссылку на внешний объект переменных из [[Scope]].
* При поиске переменных он осуществляется сначала в текущем объекте переменных, а потом – по этой ссылке.

Выглядит настолько просто, что непонятно – зачем вообще говорить об этом [[Scope]], об объектах переменных. Сказали бы: «Функция читает переменные снаружи» – и всё. Но знание этих деталей позволит нам легко объяснить и понять более сложные ситуации, с которыми мы столкнёмся далее.

## [Всегда текущее значение](http://learn.javascript.ru/closures" \l "vsegda-tekuschee-znachenie)

Значение переменной из внешней области берётся всегда текущее. Оно может быть уже не то, что было на момент создания функции.

Например, в коде ниже функция sayHi берёт phrase из внешней области:

var phrase = 'Привет';

function sayHi(name) {

alert(phrase + ', ' + name);

}

sayHi('Вася'); // Привет, Вася (\*)

phrase = 'Пока';

sayHi('Вася'); // Пока, Вася (\*\*)

На момент первого запуска (\*), переменная phrase имела значение 'Привет', а ко второму (\*\*) изменила его на 'Пока'.

Это естественно, ведь для доступа к внешней переменной функция по ссылке [[Scope]] обращается во внешний объект переменных и берёт то значение, которое там есть на момент обращения.

## [Вложенные функции](http://learn.javascript.ru/closures" \l "vlozhennye-funktsii)

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

К примеру, вложенная функция может помочь лучше организовать код:

function sayHiBye(firstName, lastName) {

alert( "Привет, " + getFullName() );

alert( "Пока, " + getFullName() );

function getFullName() {

return firstName + " " + lastName;

}

}

sayHiBye("Вася", "Пупкин"); // Привет, Вася Пупкин ; Пока, Вася Пупкин

Здесь, для удобства, создана вспомогательная функция getFullName().

Вложенные функции получают [[Scope]] так же, как и глобальные. В нашем случае:

getFullName.[[Scope]] = объект переменных текущего запуска sayHiBye

Благодаря этому getFullName() получает снаружи firstName и lastName.

Заметим, что если переменная не найдена во внешнем объекте переменных, то она ищется в ещё более внешнем (через [[Scope]] внешней функции), то есть, такой пример тоже будет работать:

var phrase = 'Привет';

function say() {

function go() {

alert( phrase ); // найдёт переменную снаружи

}

go();

}

say();

## [Возврат функции](http://learn.javascript.ru/closures" \l "vozvrat-funktsii)

Рассмотрим более «продвинутый» вариант, при котором внутри одной функции создаётся другая и возвращается в качестве результата.

В разработке интерфейсов это совершенно стандартный приём, функция затем может назначаться как обработчик действий посетителя.

Здесь мы будем создавать функцию-счётчик, которая считает свои вызовы и возвращает их текущее число.

В примере ниже makeCounter создает такую функцию:

function makeCounter() {

var currentCount = 1;

return function() { // (\*\*)

return currentCount++;

};

}

var counter = makeCounter(); // (\*)

// каждый вызов увеличивает счётчик и возвращает результат

alert( counter() ); // 1

alert( counter() ); // 2

alert( counter() ); // 3

// создать другой счётчик, он будет независим от первого

var counter2 = makeCounter();

alert( counter2() ); // 1

Как видно, мы получили два независимых счётчика counter и counter2, каждый из которых незаметным снаружи образом сохраняет текущее количество вызовов.

Где? Конечно, во внешней переменной currentCount, которая у каждого счётчика своя.

Если подробнее описать происходящее:

1. В строке (\*) запускается makeCounter(). При этом создаётся LexicalEnvironment для переменных текущего вызова. В функции есть одна переменная var currentCount, которая станет свойством этого объекта. Она изначально инициализуется в undefined, затем, в процессе выполнения, получит значение 1:

function makeCounter() {

// LexicalEnvironment = { currentCount: undefined }

var currentCount = 1;

// LexicalEnvironment = { currentCount: 1 }

return function() { // [[Scope]] -> LexicalEnvironment (\*\*)

return currentCount++;

};

}

var counter = makeCounter(); // (\*)

1. В процессе выполнения makeCounter() создаёт функцию в строке (\*\*). При создании эта функция получает внутреннее свойство [[Scope]] со ссылкой на текущий LexicalEnvironment.
2. Далее вызов makeCounter() завершается и функция (\*\*) возвращается и сохраняется во внешней переменной counter (\*).

На этом создание «счётчика» завершено.

Итоговым значением, записанным в переменную counter, является функция:

function() { // [[Scope]] -> {currentCount: 1}

return currentCount++;

};

Возвращённая из makeCounter() функция counter помнит (через [[Scope]]) о том, в каком окружении была создана.

Это и используется для хранения текущего значения счётчика.

Далее, когда-нибудь, функция counter будет вызвана. Мы не знаем, когда это произойдёт. Может быть, прямо сейчас, но, вообще говоря, совсем не факт.

Эта функция состоит из одной строки: return currentCount++, ни переменных ни параметров в ней нет, поэтому её собственный объект переменных, для краткости назовём его LE – будет пуст.

Однако, у неё есть свойство [[Scope]], которое указывает на внешнее окружение. Чтобы увеличить и вернуть currentCount, интерпретатор ищет в текущем объекте переменных LE, не находит, затем идёт во внешний объект, там находит, изменяет и возвращает новое значение:

function makeCounter() {

var currentCount = 1;

return function() {

return currentCount++;

};

}

var counter = makeCounter(); // [[Scope]] -> {currentCount: 1}

alert( counter() ); // 1, [[Scope]] -> {currentCount: 1}

alert( counter() ); // 2, [[Scope]] -> {currentCount: 2}

alert( counter() ); // 3, [[Scope]] -> {currentCount: 3}

**Переменную во внешней области видимости можно не только читать, но и изменять.**

В примере выше было создано несколько счётчиков. Все они взаимно независимы:

var counter = makeCounter();

var counter2 = makeCounter();

alert( counter() ); // 1

alert( counter() ); // 2

alert( counter() ); // 3

alert( counter2() ); // 1, счётчики независимы

Они независимы, потому что при каждом запуске makeCounter создаётся свой объект переменных LexicalEnvironment, со своим свойством currentCount, на который новый счётчик получит ссылку [[Scope]].

## [Свойства функции](http://learn.javascript.ru/closures" \l "svoystva-funktsii)

Функция в JavaScript является объектом, поэтому можно присваивать свойства прямо к ней, вот так:

function f() {}

f.test = 5;

alert( f.test );

Свойства функции не стоит путать с переменными и параметрами. Они совершенно никак не связаны. Переменные доступны только внутри функции, они создаются в процессе её выполнения. Это – использование функции «как функции».

А свойство у функции – доступно отовсюду и всегда. Это – использование функции «как объекта».

Если хочется привязать значение к функции, то можно им воспользоваться вместо внешних переменных.

В качестве демонстрации, перепишем пример со счётчиком:

function makeCounter() {

function counter() {

return counter.currentCount++;

};

counter.currentCount = 1;

return counter;

}

var counter = makeCounter();

alert( counter() ); // 1

alert( counter() ); // 2

При запуске пример работает также.

Принципиальная разница – во внутренней механике и в том, что свойство функции, в отличие от переменной из замыкания – общедоступно, к нему имеет доступ любой, у кого есть объект функции.

Например, можно взять и поменять счётчик из внешнего кода:

var counter = makeCounter();

alert( counter() ); // 1

counter.currentCount = 5;

alert( counter() ); // 5

**Статические переменные**

Иногда свойства, привязанные к функции, называют «статическими переменными».

В некоторых языках программирования можно объявлять переменную, которая сохраняет значение между вызовами функции. В JavaScript ближайший аналог – такое вот свойство функции.

**[Итого: замыкания](http://learn.javascript.ru/closures" \l "itogo-zamykaniya)**

[Замыкание](http://en.wikipedia.org/wiki/Closure_(computer_science)) – это функция вместе со всеми внешними переменными, которые ей доступны.

Таково стандартное определение, которое есть в Wikipedia и большинстве серьёзных источников по программированию. То есть, замыкание – это функция + внешние переменные.

Тем не менее, в JavaScript есть небольшая терминологическая особенность.

**Обычно, говоря «замыкание функции», подразумевают не саму эту функцию, а именно внешние переменные.**

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

**Что это такое – «понимать замыкания?»**

Иногда говорят «Вася молодец, понимает замыкания!». Что это такое – «понимать замыкания», какой смысл обычно вкладывают в эти слова?

«Понимать замыкания» в JavaScript означает понимать следующие вещи:

1. Все переменные и параметры функций являются свойствами объекта переменных LexicalEnvironment. Каждый запуск функции создает новый такой объект. На верхнем уровне им является «глобальный объект», в браузере – window.
2. При создании функция получает системное свойство [[Scope]], которое ссылается на LexicalEnvironment, в котором она была создана.
3. При вызове функции, куда бы её ни передали в коде – она будет искать переменные сначала у себя, а затем во внешних LexicalEnvironment с места своего «рождения».

В следующих главах мы углубим это понимание дополнительными примерами, а также рассмотрим, что происходит с памятью.