Современный учебник JavaScript

© Илья Кантор

Сборка от 27 апреля 2014 для печати

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

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:

```
function sayHi() {
  alert('Привет');
}

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:

```
function sayHi() { }

sayHi.test = 5;

alert(sayHi.test); // 5
```

Используется эта возможность достаточно редко.

Копирование функций

Функцию можно скопировать в другую переменную.

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

var func = sayHi;

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:

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

var func = sayHi;

func('Вася'); // выведет 'Привет, Вася'

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 [1] 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:

```
function func() { alert(1); }

var g = func; // скопировали

func = null;  // поменяли значение

g(); // работает, теперь функция в g, а в func ничего нет
func(); // вызываем null()? ошибка!
```

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

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

```
sayHi("Вася");

function sayHi(name) {
  alert("Привет, " + name);
}
```

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:

```
var age = 20;

if (age >= 18) {
  function sayHi() {  alert('Прошу вас!');  }
} else {
  function sayHi() {  alert('До 18 нельзя'); }
}

sayHi();
```

Какой вызов сработает?

Чтобы это понять — вспомним, как работают функции.

1. `Function Declaration` обрабатываются до выполнения первой строчки 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:

   ```
   function sayHi() {  alert('Прошу вас!');  }
   function sayHi() {  alert('До 18 нельзя'); }

   var age = 20;

   if (age >= 18) {

   } else {

   }

   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:

```
var f = function(параметры) {
   // тело функции
};
```

Например:

```
var sayHi = function(person) {
    alert("Привет, " + person);
};

sayHi('Вася');
```

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

Например, вполне допустимо такое объявление массива:

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

var fun = arr[2];
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.

```
sayHi(); // <-- работать не будет, функции еще нет

var sayHi = function() {  alert(1)  };
```

… А вот так — будет:

```
var sayHi = function() {  alert(1)  };

sayHi(); // 1
```

Благодаря этому свойству `Function Expression` можно (и даже нужно) использовать для условного объявления функции:

```
var age = prompt('Сколько вам лет?');
var sayHi;

if (age >= 18) {
  sayHi = function() {  alert('Вход разрешен');  }
} else {
  sayHi = function() {  alert('Извините, вы слишком молоды');  }
}

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:

```
(function() {

  var a = 1 , b = 2; // переменные для нашего скрипта

  // код скрипта

})();
```

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:

```
function() {
 // syntax error
}();
```

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:

```
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');
}();
```

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:

```
// функция объявлена и тут же вызвана
var res = function(a,b) { return a+b }(2,2);
alert(res); // 4
```

Функция здесь создаётся как часть выражения присваивания, поэтому является `Function Expression` и может быть вызвана «на месте». При этом, так как сама функция нигде не сохраняется, то она исчезнет, выполнившись, останется только её результат.

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.

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 строчки 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`

Как общий совет — предпочитайте `Function Declaration`.

Сравните по читаемости:

```
// Function Expression
var f = function() { ... }

// Function Declaration
function f() { ... }
```

`Function Declaration` короче и лучше читается. 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.

Именованные функциональные выражения

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

```
function f() { alert(1); };
g = f;
f = 0;

g(); // сменили имя f на g!
```

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

Named Function Expression

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 IE8-.

Например:

```
var f = function sayHi(name) {
  alert(sayHi); // изнутри функции - видно (выведет код функции)
};

alert(sayHi); // снаружи - не видно (ошибка: undefined variable 'sayHi')
```

Ещё одно принципиальное отличие имени от обычной переменной заключается в том, что его нельзя перезаписать:

```
var test = function sayHi(name) {
  sayHi = "тест";
  alert(sayHi); // function... (перезапись не удалась)
};

test();
```

В режиме `use strict` код выше выдал бы ошибку.

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 [2].

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

Пример использования

NFE используется в первую очередь в тех ситуациях, когда функцию нужно передавать в другое место кода или перемещать из одной переменной в другую.

Внутреннее имя позволяет функции надёжно обращаться к самой себе, где бы она ни находилась.

Вспомним, к примеру, функцию-факториал из задачи Вычислить факториал [3]:

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

alert( f(5) ); // 120
```

Попробуем перенести её в другую переменную:

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

var g = f;
f = null;

alert( g(5) ); // ошибка при выполнении!
```

Ошибка возникла потому что функция из своего кода обращается к своему старому имени `f`. А этой функции уже нет, `f = null`.

Для того, чтобы функция всегда надёжно работала, объявим её как Named Function Expression:

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

var g = f;  // скопировали ссылку на функцию-факториал в g
f = null;

alert( g(5) ); // 120, работает!
```

В браузере IE8- 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:

```
var f = function factorial(n) { /*...*/ };

// в IE8- false
// в остальных браузерах ошибка, т.к. имя factorial не видно
alert(f === factorial);
```

Все остальные браузеры полностью поддерживают именованные функциональные выражения.

Итого

Если функция задана как Function Expression, её можно дать имя. Оно будет доступно только внутри функции (кроме IE8-) и предназначено для надёжного рекурсивного вызова функции, даже если она записана в другую переменную.

Далее в учебнике мы посмотрим ещё примеры применения Named Function Expression для удобства разработки.

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:

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

Создать переменную можно и явным присваиванием в `window`:

```
window.a = 5;
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 [4], 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 строчки кода.

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

```
// По окончании инициализации, до выполнения кода:
// window = { f: function, a: undefined, g: undefined }

var a = 5; // при инициализации даёт: window.a=undefined

function f(arg) { /*...*/ }  // при инициализации даёт: window.f = function

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:

```
alert("a" in window); // true,  т.к. есть свойство window.a
alert(a); // равно undefined,  присваивание будет выполнено далее
alert(f); // function ...,  готовая к выполнению функция
alert(g); // undefined, т.к. это переменная, а не Function Declaration

var a = 5;
function f() { /*...*/ }
var g = function() { /*...*/ };
```

Присвоение переменной без объявления

В старом стандарте JavaScript переменную можно было создать и без объявления `var`:

```
a = 5;

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:

```
alert(a); // undefined

var a = 5;
```

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

```
alert(a); // error, a is not defined

a = 5;
```

Вообще, рекомендуется всегда объявлять переменные через `var`.

В современном стандарте присваивание без `var` вызовет ошибку:

```
'use strict';
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:

```
var i;
{
  i = 5;
}
```

…И внутри него:

```
i = 5;
{
  var i;
}
```

Также нет разницы между объявлением в цикле и вне его:

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

Идентичный по функциональности код:

```
var i;
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:

```
var i = 10;

for (var i=0; i<20; i++) {
  ...
}

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:

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

```
<script>
// рекурсия через функцию, явно записанную в window
window.recurse = function(times) {
  if (times !== 0) recurse(times-1);
}

recurse(13);
</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 выполнятся позже, когда выполнение дойдет до них.

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:

```
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');
   ```

2. 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');
   ```

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:

```
var a = 5;

function f() {
  alert(a);  // 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:

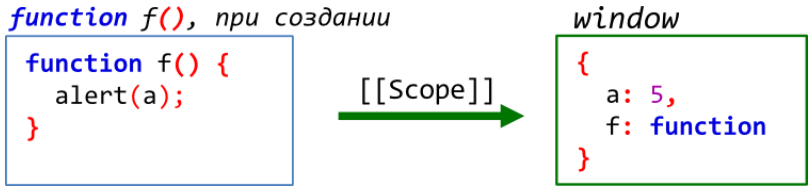
   ```
   var a = 5

   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]]`:

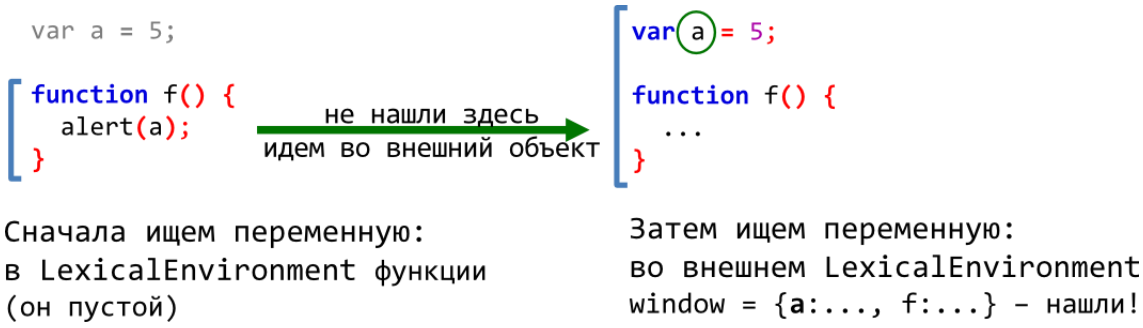


   ...И использует его при создании объекта переменных для функции.

   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.

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`:

```
LexicalEnvironment = {
  person: переданный аргумент,
  message: undefined,
  getHello: function...,
  makeMessage: function...
}
```

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:

```
function sayHi(person) {
  ...
  function makeMessage(person) {
    return getHello(person.age) + ', ' + person.name;
  }
}
```

Станет:

```
function sayHi(person) {
  ...
  function makeMessage() { // убрали аргумент
    // переменная person будет взята из внешнего объекта переменных
    return getHello(person.age) + ', ' + person.name;
  }
}
```

Вложенную функцию можно возвратить.

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

```
function sayHi(person) {

  return function() { // (*)
    var message = makeMessage(person); // (**)
    alert( message );
  };

  // ----- вспомогательные функции -----

  function getHello(age) {
    return age >= 18 ? 'Здравствуйте' : 'Привет';
  }

  function makeMessage() {
    return getHello(person.age) + ', ' + person.name;
  }
}

var sayHiPete = sayHi({ name: 'Петька', age: 17 });
var sayHiOther = sayHi({ name: 'Василий Иванович', age: 35 });

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 `makeMessage` при вызове в строке `(**)` будет взята из внешнего объекта переменных.

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

Замыканием [5] 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 [6].

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:

```
function f() {
  var value = 123;
  function g() { } // g видна только изнутри
}

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:

```
function f() {
  var a = Math.random();

  function g() { }

  return g;
}

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:

```
function f() {
  var a = Math.random();

  return function() { };
}

// 3 функции,
// каждая ссылается на соответствующий объект LexicalEnvironment = {a: ...}
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:

```
function f() {
  var a = Math.random();

  function g() { }

  return g;
}

var g = f();  // функция g жива
// а значит в памяти остается соответствующий объект переменных

g = null;     // ..а вот теперь память будет очищена
```

[[Scope]] для new Function

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

Существует ещё один способ создания функции, о котором мы не говорили раньше, поскольку он используется очень редко. Он выглядит так:

```
var sum = new Function('a,b', ' return a+b; ');

var result = sum(1,2);
alert(result); // 3
```

То есть, функция создаётся вызовом new Function(params, code):

params
: Параметры функции через запятую в виде строки.

code
: Код функции в виде строки.

Этот способ используется очень редко, но в отдельных случаях бывает весьма полезен, так как позволяет конструировать функцию во время выполнения программы, к примеру из данных, полученных с сервера или от пользователя.

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 замыкании, 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:

```
function makeCounter() {
  var currentCount = 0;

  return function(newCount) {
    if (newCount !== undefined) { // есть аргумент?
      currentCount = +newCount; // сделаем его новым значением счётчика
      // вернём текущее значение, счётчик всегда возвращает его (это удобно)
      return currentCount;
    }

    currentCount++;
    return currentCount;
  };
}

var counter = makeCounter();

alert( counter() ); // 1
alert( counter(3) ); // 3
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 возвратить полноценный объект с функциями управления счётчиком:

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

```
function makeCounter() {
  var currentCount = 0;

  return {
    getNext: function() {
      return ++currentCount;
    },

    set: function(value) {
      currentCount = value;
    },

    reset: function() {
      currentCount = 0;
    }
  };
}

var counter = makeCounter();

alert( counter.getNext() ); // 1
alert( counter.getNext() ); // 2

counter.reset();
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:

```
function makeCounter() {
  var currentCount = 0;

  // возвращаемся к функции
  function counter() {
    return ++currentCount;
  }

  // ...и добавляем ей методы!
  counter.set = function(value) {
    currentCount = value;
  };

  counter.reset = function() {
    currentCount = 0;
  };

  return counter;
}

var counter = makeCounter();

alert( counter() ); // 1
alert( counter() ); // 2

counter.reset();
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. Внутрь этой функции пишутся локальные переменные и функции, которые пользователю модуля не нужны, но нужны самому модулю:

   ```
   (function() {
     var count = 0;

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

   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`:

```
var func = (function() {
  var count = 0;
  function helper() { ... }

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

   …Если функций много — можно присвоить их напрямую в `window`:

```
(function() {

  function helper() { ... }

  window.createMenu = function() { ... };
  window.createDialog = function() { ... };

})();
```

   Или, что ещё лучше, вернуть объект с ними:

```
var MyLibrary = (function() {

  function helper() { ... }

  return {
    createMenu: function() { ... },
    createDialog: function() { ... }
  };

})();

// использование
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 переменных.

Задачи на понимание замыканий

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 свойств 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.

Конструкция "with"

Конструкция `with` позволяет использовать для области видимости произвольный объект.

В современном JavaScript от этой конструкции отказались, в строгом режиме она не работает, 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`:

```
var a = 5;

var obj = { a : 10 };

with(obj) {
  alert(a); // 10, из obj
}
```

Попробуем получить переменную, которой в `obj` нет:

```
var b = 1;

var obj = { a : 10 };

with(obj) {
  alert(b); // 1, из window
}
```

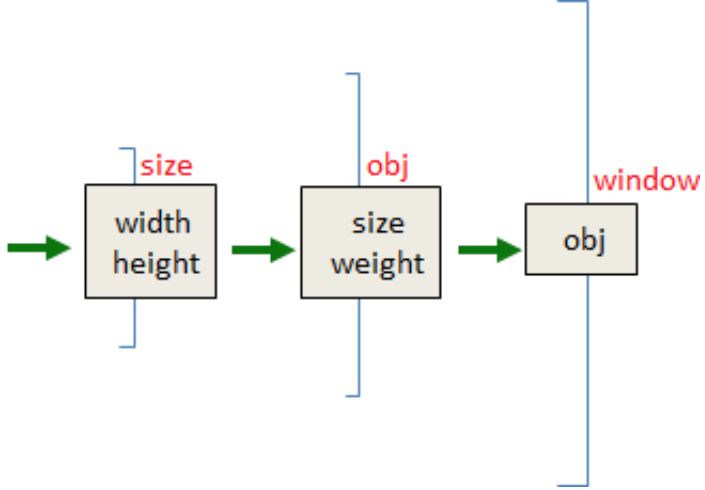
Здесь интерпретатор сначала проверяет наличие `obj.b`, не находит и идет вне `with`.

Особенно забавно выглядит применение вложенных `with`:

```
var obj = {
  weight: 10,
  size: {
    width: 5,
    height: 7
  }
};

with(obj) {
  with(size) { // size будет взят из obj
    alert( width*height / weight ); // width,height из size, weight из obj
  }
}
```

Свойства из разных объектов используются как обычные переменные… Магия! Порядок поиска переменных в выделенном коде:
`size => obj => window`

```
…
with (obj) {
  …
  with (size) {
    …
    alert( width*height / weight )
    …
  }
  …
}
…
```



Изменения переменной

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

Например:

```
var obj = { a : 10 }

with(obj) {
  a = 20;
}
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:

   ```
   var obj = { weight: 10 };

   with(obj) {
     weight = 20; // (1)
     size = 35;   // (2)
   }

   alert(obj.size);
   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`.

2. Еще одна причина — **алгоритмы сжатия JavaScript не любят `with`**. Перед выкладкой на сервер JavaScript сжимают. Для этого есть много инструментов, например Closure Compiler [7] и UglifyJS [8]. Если вкратце — они либо сжимают код с `with` с ошибками, либо оставляют его частично несжатым.
3. 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.

```
var i = 0;

function fast() {
  i++;
}

function slow() {
  with(i) {}
  i++;
}


var time = new Date();
while(i < 1000000) fast();
alert(new Date - time);

var time = new Date();
i=0;
while(i < 1000000) slow();
alert(new Date - time);
```

Замена `with`

Вместо `with` рекомендуется использовать временную переменную, например:

```
/* вместо
with(elem.style) {
  top = '10px';
  left = '20px';
}
*/

var s = elem.style;

s.top = '10px';
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.

Решения задач

Решение задачи: Проверка на NFE

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

```
// обычное объявление функции (Function Declaration)
function g() { return 1; };

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.

```
// Named Function Expression!
(function g() { return 1; });

alert(g);  // Ошибка!
```

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

Решение задачи: Window и переменная

Ответ: `1`.

```
if ("a" in window) {
    var a = 1;
}
alert(a);
```

Посмотрим, почему.

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

```
// window = {a:undefined}

if ("a" in window) { // в if видно что window.a уже есть
    var a = 1; // поэтому эта строка сработает
}
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.

```
if ("a" in window) {
    a = 1;
}
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`.

```
if ("a" in window) {
    a = 1;
}
var a;

alert(a); // 1
```

Решение задачи: Функция и переменная

Ответ: `5`.

```
var a = 5;

function a() { }

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` обработается 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`.

```
function test() {

  alert(window);

  var window = 5;

  alert(window);
}

test();
```

Директива `var` обработается 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:

```
var a = 5

(function() {
  alert(a)
})()
```

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

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

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

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

Если точку с запятой поставить, все будет хорошо:

```
var a = 5;

(function() {
  alert(a)
})()
```

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:

```
function sum(a) {

  return function(b) {
    return a + b;  // возьмет a из внешнего LexicalEnvironment
  };

}

alert( sum(1)(2) );
alert( sum(5)(-1) );
```

Решение задачи: Фильтрация через функцию

Функция фильтрации

```
function filter(arr, func) {
  var result = [];

  for(var i=0; i<arr.length; i++) {
    var val  = arr[i];
    if (func(val)) {
      result.push(val);
    }
  }

  return result;
}

var arr = [1, 2, 3, 4, 5, 6, 7];

alert( filter(arr, function(a) { return a % 2 == 0; }) ); // 2, 4, 6
```

Фильтр inBetween

```
function filter(arr, func) {
  var result = [];

  for(var i=0; i<arr.length; i++) {
    var val  = arr[i];
    if (func(val)) {
      result.push(val);
    }
  }

  return result;
}

function inBetween(a, b) {
  return function(x) {
    return x >=a && x <= b;
  };
}

var arr = [1, 2, 3, 4, 5, 6, 7];
alert( filter(arr, inBetween(3,6)) ); // 3,4,5,6
```

Фильтр inArray

```
function filter(arr, func) {
  var result = [];

  for(var i=0; i<arr.length; i++) {
    var val  = arr[i];
    if (func(val)) {
      result.push(val);
    }
  }

  return result;
}

function inArray(arr) {
  return function(x) {
    return arr.indexOf(x) != -1;
  };
}

var arr = [1, 2, 3, 4, 5, 6, 7];
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. Создаёт 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:

   ```
   shooters = [
     function () { alert(i); },
     function () { alert(i); },
     function () { alert(i); },
     function () { alert(i); },
     function () { alert(i); },
     function () { alert(i); },
     function () { alert(i); },
     function () { alert(i); },
     function () { alert(i); },
     function () { alert(i); }
   ];
   ```

   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:**

```
function makeArmy() {

  var shooters = [];

  for(var i=0; i<10; i++) {

    var shooter = function me() {
      alert( me.i );
    };
    shooter.i = i;

    shooters.push(shooter);
  }

  return shooters;
}

var army = makeArmy();

army[0](); // 0
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:

```
...
var shooter = function me() {
  alert( me.i );
};
...
```

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

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

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`**:

```
function makeArmy() {

  var shooters = [];

  for(var i=0; i<10; i++) {

    var shooter = (function(x) {

      return function() {
        alert( x );
      };

    })(i);

    shooters.push(shooter);
  }

  return shooters;
}

var army = makeArmy();

army[0](); // 0
army[1](); // 1
```

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

```
var shooter = (function(x) {
  return function() {
    alert( x );
  };
})(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:

```
var shooter = (function(i) {
  return function() {
    alert( i );
  };
})(i);
```

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

```
var shooter = function(i) { // без скобок вокруг function(i)
  return function() {
    alert( i );
  };
}(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:

```
function makeCounter() {
  return function f() {
    if (!f.currentCount) {
      f.currentCount = 0;
    }

    return ++f.currentCount;
  };
}

var c1 = makeCounter();
var c2 = makeCounter();

alert( c1() ); // 1
alert( c2() ); // 1
alert( c1() ); // 2
```

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

```
var counter = makeCounter();

counter();
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:

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

var obj = {
  f: function() { alert(2) }
};

with(obj) {
  f();
}
```

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:

```
var a = 1;
var b;

var obj = { b: 2 }

with(obj) {
  alert( a + b );
}
```

Ссылки

1. Другие варианты http://learn.javascript.ru/function-declaration-expression#function-expression
2. В отдельной главе http://learn.javascript.ru/arguments-pseudoarray#arguments-callee
3. Вычислить факториал http://learn.javascript.ru/task/vychislit-faktorial
4. Function Declaration http://learn.javascript.ru/function-declaration-expression
5. Замыканием http://en.wikipedia.org/wiki/Closure_(computer_science)
6. Управление памятью в JS и DOM http://learn.javascript.ru/memory-management
7. Closure Compiler http://code.google.com/intl/ru-RU/closure/compiler/
8. UglifyJS https://github.com/mishoo/UglifyJS