**Замыкания в JavaScript**

[Веб-разработка](http://habrahabr.ru/hub/webdev/)\*

Если вы используете JavaScript, но при этом так до конца и не разобрались, что же это за чудная штука такая — [замыкания](http://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BC%D1%8B%D0%BA%D0%B0%D0%BD%D0%B8%D0%B5_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29), и зачем она нужна — эта статья для вас.  
  
Как известно, в JavaScript областью видимости локальных переменных (объявляемых словом var) является тело функции, внутри которой они определены.  
  
Если вы объявляете функцию внутри другой функции, первая получает доступ к переменным и аргументам последней:

function outerFn(myArg) {  
   var myVar;  
   function innerFn() {  
      //имеет доступ к myVar и myArg  
   }  
}

При этом, такие переменные *продолжают существовать и остаются доступными внутренней функцией* даже после того, как внешняя функция, в которой они определены, была исполнена.   
  
Рассмотрим пример — функцию, возвращающую кол-во собственных вызовов:

function createCounter() {  
   var numberOfCalls = 0;  
   return function() {  
      return ++numberOfCalls;  
   }  
}  
var fn = createCounter();  
fn(); //1  
fn(); //2  
fn(); //3

В данном примере функция, возвращаемая createCounter, использует переменную numberOfCalls, которая сохраняет нужное значение между ее вызовами (вместо того, чтобы сразу прекратить своё существование с возвратом createCounter).  
  
Именно за эти свойства такие «вложенные» функции в JavaScript называют *замыканиями* (термином, пришедшим из функциональных языков программирования) — они «замыкают» на себя переменные и аргументы функции, внутри которой определены.

**Применение замыканий**

Упростим немножко пример выше — уберём необходимость отдельно вызывать функцию createCounter, сделав ее аномимной и вызвав сразу же после ее объявления:

var fn = (function() {  
   var numberOfCalls = 0;  
   return function() {  
      return ++ numberOfCalls;  
   }  
})();

Такая конструкция позволила нам привязать к функции данные, сохраняющиеся между ее вызовами — это одно из применений замыканий. Иными словами, с помощью них мы можем создавать функции, имеющие своё изменяемое состояние.  
  
Другое хорошее применение замыканий — создание функций, в свою очередь тоже создающих функции — то, что некоторые назвали бы приёмом т.н. метапрограммирования. Например:

var createHelloFunction = function(name) {  
   return function() {  
      alert('Hello, ' + name);  
   }  
}  
var sayHelloHabrahabr = createHelloFunction('Habrahabr');  
sayHelloHabrahabr(); //alerts «Hello, Habrahabr»

Благодаря замыканию *возвращаемая функция* «запоминает» параметры, переданные *функции создающей*, что нам и нужно для подобного рода вещей.  
  
Похожая ситуация возникает, когда мы внутреннюю функцию не возвращаем, а вешаем на какое-либо событие — поскольку событие возникает уже после того, как исполнилась функция, замыкание опять же помогает не потерять переданные при создании обработчика данные.  
  
Рассмотрим чуть более сложный пример — метод, привязывающий функцию к определённому контексту (т.е. объекту, на который в ней будет указывать слово this).

Function.prototype.bind = function(context) {  
   var fn = this;  
   return function() {  
      return fn.apply(context, arguments);  
   };  
}  
var HelloPage = {  
   name: 'Habrahabr',  
   init: function() {  
      alert('Hello, ' + this.name);  
   }  
}  
//window.onload = HelloPage.init; //алертнул бы undefined, т.к. this указывало бы на window  
window.onload = HelloPage.init.bind(HelloPage); //вот теперь всё работает

В этом примере с помощью замыканий функция, вощвращаемая bind'ом, запоминает в себе начальную функцию и присваиваемый ей контекст.  
  
Следующее, принципиально иное применение замыканий — защита данных (инкапсуляция). Рассмотрим следующую конструкцию:

(function() {  
   …  
})();

Очевидно, внутри замыкания мы имеем доступ ко всем внешним данным, но при этом оно имеет и собственные. Благодаря этому мы можем окружать части кода подобной конструкцией с целью закрыть попавшие внутрь локальные переменные от доступа снаружи. (Один из примеров ее использования вы можете увидеть в исходном коде библиотеки jQuery, которая окружает замыканием весь свой код, чтобы не выводить за его пределы нужные только ей переменные).  
  
Есть, правда, одна связанная с таким применением ловушка — внутри замыкания теряется значение слова this за его пределами. Решается она следующим образом:

(function() {  
   //вышестоящее this сохранится  
}).call(this);

Рассмотрим еще один приём из той же серии. Повсеместно популяризовали его разработчики фреймворка Yahoo UI, назвав его «Module Pattern» и написав о нём [целую статью в официальном блоге](http://yuiblog.com/blog/2007/06/12/module-pattern/).  
  
Пускай у нас есть объект ([синглтон](http://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D0%B3%D0%BB%D1%82%D0%BE%D0%BD)), содержащий какие-либо методы и свойства:

var MyModule = {  
   name: 'Habrahabr',  
   sayPreved: function(name) {  
      alert('PREVED ' + name.toUpperCase())  
   },     
   sayPrevedToHabrahabr: function() {  
      this.sayPreved(this.name);  
   }  
}  
MyModule.sayPrevedToHabrahabr();

С помощью замыкания мы можем сделать методы и свойства, которые вне объекта не используютя, приватными (т.е. доступными только ему):

var MyModule = (function() {  
   var name = 'Habrahabr';  
   function sayPreved() {  
      alert('PREVED ' + name.toUpperCase());  
   }  
   return {  
      sayPrevedToHabrahabr: function() {  
         sayPreved(name);  
      }  
   }  
})();  
MyModule.sayPrevedToHabrahabr(); //alerts «PREVED Habrahabr»

Напоследок хочу описать распространённую ошибку, которая многих вгоняет в ступор в случае незнания того, как работают замыкания.   
  
Пускай у нас есть массив ссылок, и наша задача — сделать так, чтобы при клике на каждую выводился алертом ее порядковый номер. Первое решение, что приходит в голову, выглядит так:

for (var i = 0; i < links.length; i++) {  
   links[i].onclick = function() {  
      alert(i);  
   }  
}

На деле же оказывается, что при клике на любую ссылку выводится одно и то же число — значение links.length. Почему так происходит? В связи с замыканием объявленная вспомогательная переменная i продолжает существовать, при чём и в тот момент, когда мы кликаем по ссылке. Поскольку к тому времени цикл уже прошёл, i остаётся равным кол-ву ссылок — это значение мы и видим при кликах.  
  
Решается эта проблема следующим образом:

for (var i = 0; i < links.length; i++) {  
   (function(i) {  
      links[i].onclick = function() {  
         alert(i);  
      }  
   })(i);  
}

Здесь с помощью еще одного замыкания мы «затеняем» переменную i, создавая ее копию в его локальной области видимости на каждом шаге цикла. Благодаря этому всё теперь работает как задумывалось.