**Привязка контекста и карринг: "bind"**

Функции в JavaScript никак не привязаны к своему контексту this, с одной стороны, здорово – это позволяет быть максимально гибкими, одалживать методы и так далее.

Но с другой стороны – в некоторых случаях контекст может быть потерян. То есть мы вроде как вызываем метод объекта, а на самом деле он получает this = undefined.

Такая ситуация является типичной для начинающих разработчиков, но бывает и у «зубров» тоже. Конечно, «зубры» при этом знают, что с ней делать.

**[Пример потери контекста](http://learn.javascript.ru/bind" \l "primer-poteri-konteksta)**

В браузере есть встроенная функция setTimeout(func, ms), которая вызывает выполнение функции func через ms миллисекунд (=1/1000 секунды).

Мы подробно остановимся на ней и её тонкостях позже, в главе [setTimeout и setInterval](http://learn.javascript.ru/settimeout-setinterval), а пока просто посмотрим пример.

Этот код выведет «Привет» через 1000 мс, то есть 1 секунду:

setTimeout(function() {

alert( "Привет" );

}, 1000);

Попробуем сделать то же самое с методом объекта, следующий код должен выводить имя пользователя через 1 секунду:

var user = {

firstName: "Вася",

sayHi: function() {

alert( this.firstName );

}

};

setTimeout(user.sayHi, 1000); // undefined (не Вася!)

При запуске кода выше через секунду выводится вовсе не "Вася", а undefined!

Это произошло потому, что в примере выше setTimeout получил функцию user.sayHi, но не её контекст. То есть, последняя строчка аналогична двум таким:

var f = user.sayHi;

setTimeout(f, 1000); // контекст user потеряли

Ситуация довольно типична – мы хотим передать метод объекта куда-то в другое место кода, откуда он потом может быть вызван. Как бы прикрепить к нему контекст, желательно, с минимумом плясок с бубном и при этом надёжно?

Есть несколько способов решения, среди которых мы, в зависимости от ситуации, можем выбирать.

**[Решение 1: сделать обёртку](http://learn.javascript.ru/bind" \l "reshenie-1-sdelat-obyortku)**

Самый простой вариант решения – это обернуть вызов в анонимную функцию:

var user = {

firstName: "Вася",

sayHi: function() {

alert( this.firstName );

}

};

setTimeout(function() {

user.sayHi(); // Вася

}, 1000);

Теперь код работает, так как user достаётся из замыкания.

Это решение также позволяет передать дополнительные аргументы:

var user = {

firstName: "Вася",

sayHi: function(who) {

alert( this.firstName + ": Привет, " + who );

}

};

setTimeout(function() {

user.sayHi("Петя"); // Вася: Привет, Петя

}, 1000);

Но тут же появляется и уязвимое место в структуре кода!

А что, если до срабатывания setTimeout (ведь есть целая секунда) в переменную user будет записано другое значение? К примеру, в другом месте кода будет присвоено user=(другой пользователь)… В этом случае вызов неожиданно будет совсем не тот!

Хорошо бы гарантировать правильность контекста.

**[Решение 2: bind для привязки контекста](http://learn.javascript.ru/bind" \l "reshenie-2-bind-dlya-privyazki-konteksta)**

Напишем вспомогательную функцию bind(func, context), которая будет жёстко фиксировать контекст для func:

function bind(func, context) {

return function() { // (\*)

return func.apply(context, arguments);

};

}

Посмотрим, что она делает, как работает, на таком примере:

function f() {

alert( this );

}

var g = bind(f, "Context");

g(); // Context

То есть, bind(f, "Context") привязывает "Context" в качестве this для f.

Посмотрим, за счёт чего это происходит.

Результатом bind(f, "Context"), как видно из кода, будет анонимная функция (\*).

Вот она отдельно:

function() { // (\*)

return func.apply(context, arguments);

};

Если подставить наши конкретные аргументы, то есть f и "Context", то получится так:

function() { // (\*)

return f.apply("Context", arguments);

};

Эта функция запишется в переменную g.

Далее, если вызвать g, то вызов будет передан в f, причём f.apply("Context", arguments) передаст в качестве контекста "Context", который и будет выведен.

Если вызвать g с аргументами, то также будет работать:

function f(a, b) {

alert( this );

alert( a + b );

}

var g = bind(f, "Context");

g(1, 2); // Context, затем 3

Аргументы, которые получила g(...), передаются в f также благодаря методу .apply.

**Иными словами, в результате вызова bind(func, context) мы получаем «функцию-обёртку», которая прозрачно передаёт вызов в func, с теми же аргументами, но фиксированным контекстом context.**

Вернёмся к user.sayHi. Вариант с bind:

function bind(func, context) {

return function() {

return func.apply(context, arguments);

};

}

var user = {

firstName: "Вася",

sayHi: function() {

alert( this.firstName );

}

};

setTimeout(bind(user.sayHi, user), 1000);

Теперь всё в порядке!

Вызов bind(user.sayHi, user) возвращает такую функцию-обёртку, которая привязывает user.sayHi к контексту user. Она будет вызвана через 1000 мс.

Полученную обёртку можно вызвать и с аргументами – они пойдут в user.sayHi без изменений, фиксирован лишь контекст.

var user = {

firstName: "Вася",

sayHi: function(who) { // здесь у sayHi есть один аргумент

alert( this.firstName + ": Привет, " + who );

}

};

var sayHi = bind(user.sayHi, user);

// контекст Вася, а аргумент передаётся "как есть"

sayHi("Петя"); // Вася: Привет, Петя

sayHi("Маша"); // Вася: Привет, Маша

В примере выше продемонстрирована другая частая цель использования bind – «привязать» функцию к контексту, чтобы в дальнейшем «не таскать за собой» объект, а просто вызывать sayHi.

Результат bind можно передавать в любое место кода, вызывать как обычную функцию, он «помнит» свой контекст.

**[Решение 3: встроенный метод bind](http://learn.javascript.ru/bind" \l "bind)**

В современном JavaScript (или при подключении библиотеки [es5-shim](https://github.com/kriskowal/es5-shim) для IE8-) у функций уже есть встроенный метод [bind](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Function/bind), который мы можем использовать.

Он работает примерно так же, как bind, который описан выше.

Изменения очень небольшие:

function f(a, b) {

alert( this );

alert( a + b );

}

// вместо

// var g = bind(f, "Context");

var g = f.bind("Context");

g(1, 2); // Context, затем 3

Синтаксис встроенного bind:

var wrapper = func.bind(context[, arg1, arg2...])

**func**

Произвольная функция

**context**

Контекст, который привязывается к func

**arg1, arg2, …**

Если указаны аргументы arg1, arg2... – они будут прибавлены к каждому вызову новой функции, причем встанут *перед* теми, которые указаны при вызове.

Результат вызова func.bind(context) аналогичен вызову bind(func, context), описанному выше. То есть, wrapper – это обёртка, фиксирующая контекст и передающая вызовы в func. Также можно указать аргументы, тогда и они будут фиксированы, но об этом чуть позже.

Пример со встроенным методом bind:

var user = {

firstName: "Вася",

sayHi: function() {

alert( this.firstName );

}

};

// setTimeout( bind(user.sayHi, user), 1000 );

setTimeout(user.sayHi.bind(user), 1000); // аналог через встроенный метод

Получили простой и надёжный способ привязать контекст, причём даже встроенный в JavaScript.

Далее мы будем использовать именно встроенный метод bind.

**bind не похож на call/apply**

Методы bind и call/apply близки по синтаксису, но есть важнейшее отличие.

Методы call/apply вызывают функцию с заданным контекстом и аргументами.

А bind не вызывает функцию. Он только возвращает «обёртку», которую мы можем вызвать позже, и которая передаст вызов в исходную функцию, с привязанным контекстом.

**Привязать всё: bindAll**

Если у объекта много методов и мы планируем их активно передавать, то можно привязать контекст для них всех в цикле:

for (var prop in user) {

if (typeof user[prop] == 'function') {

user[prop] = user[prop].bind(user);

}

}

В некоторых JS-фреймворках есть даже встроенные функции для этого, например [\_.bindAll(obj)](http://lodash.com/docs#bindAll).

**[Карринг](http://learn.javascript.ru/bind" \l "karring)**

До этого мы говорили о привязке контекста. Теперь пойдём на шаг дальше. Привязывать можно не только контекст, но и аргументы. Используется это реже, но бывает полезно.

[Карринг](http://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%80%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) (currying) или *каррирование* – термин [функционального программирования](http://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%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), который означает создание новой функции путём фиксирования аргументов существующей.

Как было сказано выше, метод func.bind(context, ...) может создавать обёртку, которая фиксирует не только контекст, но и ряд аргументов функции.

Например, есть функция умножения двух чисел mul(a, b):

function mul(a, b) {

return a \* b;

};

При помощи bind создадим функцию double, удваивающую значения. Это будет вариант функции mul с фиксированным первым аргументом:

// double умножает только на два

var double = mul.bind(null, 2); // контекст фиксируем null, он не используется

alert( double(3) ); // = mul(2, 3) = 6

alert( double(4) ); // = mul(2, 4) = 8

alert( double(5) ); // = mul(2, 5) = 10

При вызове double будет передавать свои аргументы исходной функции mul после тех, которые указаны в bind, то есть в данном случае после зафиксированного первого аргумента 2.

**Говорят, что double является «частичной функцией» (partial function) от mul.**

Другая частичная функция triple утраивает значения:

var triple = mul.bind(null, 3); // контекст фиксируем null, он не используется

alert( triple(3) ); // = mul(3, 3) = 9

alert( triple(4) ); // = mul(3, 4) = 12

alert( triple(5) ); // = mul(3, 5) = 15

При помощи bind мы можем получить из функции её «частный вариант» как самостоятельную функцию и дальше передать в setTimeout или сделать с ней что-то ещё.

Наш выигрыш состоит в том, что эта самостоятельная функция, во-первых, имеет понятное имя (double, triple), а во-вторых, повторные вызовы позволяют не указывать каждый раз первый аргумент, он уже фиксирован благодаря bind.

**[Функция ask для задач](http://learn.javascript.ru/bind" \l "funktsiya-ask-dlya-zadach)**

В задачах этого раздела предполагается, что объявлена следующая «функция вопросов» ask:

function ask(question, answer, ok, fail) {

var result = prompt(question, '');

if (result.toLowerCase() == answer.toLowerCase()) ok();

else fail();

}

Её назначение – задать вопрос question и, если ответ совпадёт с answer, то запустить функцию ok(), а иначе – функцию fail().

Несмотря на внешнюю простоту, функции такого вида активно используются в реальных проектах. Конечно, они будут сложнее, вместо alert/prompt – вывод красивого JavaScript-диалога с рамочками, кнопочками и так далее, но это нам сейчас не нужно.

Пример использования:

ask("Выпустить птичку?", "да", fly, die);

function fly() {

alert( 'улетела :)' );

}

function die() {

alert( 'птичку жалко :(' );

}

**[Итого](http://learn.javascript.ru/bind" \l "itogo)**

* Функция сама по себе не запоминает контекст выполнения.
* Чтобы гарантировать правильный контекст для вызова obj.func(), нужно использовать функцию-обёртку, задать её через анонимную функцию:

setTimeout(function() {

obj.func();

})

* …Либо использовать bind:

setTimeout(obj.func.bind(obj));

* Вызов bind часто используют для привязки функции к контексту, чтобы затем присвоить её в обычную переменную и вызывать уже без явного указания объекта.
* Вызов bind также позволяет фиксировать первые аргументы функции («каррировать» её), и таким образом из общей функции получить её «частные» варианты – чтобы использовать их многократно без повтора одних и тех же аргументов каждый раз.