Объекты. Контекст вызова

Оглавление

[Введение в методы и свойства 0](#_Toc454998602)

[Преобразование объектов 4](#_Toc454998603)

[Статические свойства и методы 6](#_Toc454998604)

[Явное указание контекста 7](#_Toc454998605)

[Домашнее задание 9](#_Toc454998606)

[Используемая литература 10](#_Toc454998607)

# Введение в методы и свойства

При объявлении объекта можно указать свойство-функцию, например:

|  |
| --- |
| var user = {  name: 'Василий',  // метод  sayHi: function() {  alert( 'Привет!' );  }  };  // Вызов  user.sayHi(); |

Свойства-функции называют «методами» объектов. Их можно добавлять и удалять в любой момент, в том числе и явным присваиванием:

|  |
| --- |
| var user = {  name: 'Василий'  };  user.sayHi = function() { // присвоили метод после создания объекта  alert('Привет!');  };  // Вызов метода:  user.sayHi(); |

Для полноценной работы метод должен иметь доступ к данным объекта. В частности, вызовuser.sayHi() может захотеть вывести имя пользователя.

**Для доступа к текущему объекту из метода используется ключевое слово this**.

Значением this является объект перед «точкой», в контексте которого вызван метод, например:

|  |
| --- |
| var user = {  name: 'Василий',  sayHi: function() {  alert( this.name );  }  };  user.sayHi(); // sayHi в контексте user |

Здесь при выполнении функции user.sayHi() в this будет храниться ссылка на текущий объектuser.

Вместо this внутри sayHi можно было бы обратиться к объекту, используя переменную user:

|  |
| --- |
| ...  sayHi: function() {  alert( user.name );  }  ... |

…Однако, такое решение нестабильно. Если мы решим скопировать объект в другую переменную, например admin = user, а в переменную user записать что-то другое – обращение будет совсем не по адресу:

|  |
| --- |
| var user = {  name: 'Василий',  sayHi: function() {  alert( user.name ); // приведёт к ошибке  }  };  var admin = user;  user = null;  admin.sayHi(); // упс! внутри sayHi обращение по старому имени, ошибка! |

Использование this гарантирует, что функция работает именно с тем объектом, в контексте которого вызвана.

Через this метод может не только обратиться к любому свойству объекта, но и передать куда-то ссылку на сам объект целиком:

|  |
| --- |
| var user = {  name: 'Василий',  sayHi: function() {  showName(this); // передать текущий объект в showName  }  };  function showName(namedObj) {  alert( namedObj.name );  }  user.sayHi(); // Василий |

Любая функция может иметь в себе this. Совершенно неважно, объявлена ли она в объекте или отдельно от него.

Значение this называется *контекстом вызова* и будет определено в момент вызова функции.

Например, такая функция, объявленная без объекта, вполне допустима:

|  |
| --- |
| function sayHi() {  alert( this.firstName );  } |

Эта функция ещё не знает, каким будет this. Это выяснится при выполнении программы.

**Если одну и ту же функцию запускать в контексте разных объектов, она будет получать разный this:**

|  |
| --- |
| var user = { firstName: "Вася" };  var admin = { firstName: "Админ" };  function func() {  alert( this.firstName );  }  user.f = func;  admin.g = func;  // this равен объекту перед точкой:  user.f(); // Вася  admin.g(); // Админ  admin['g'](); // Админ (не важно, доступ к объекту через точку или квадратные скобки) |

Итак, значение this не зависит от того, как функция была создана, оно определяется исключительно в момент вызова.

Если функция использует this – это подразумевает работу с объектом. Но и прямой вызов func()технически возможен.

Как правило, такая ситуация возникает при ошибке в разработке.

При этом this получает значение window, глобального объекта:

|  |
| --- |
| function func() {  alert( this ); // выведет [object Window] или [object global]  }  func(); |

Таково поведение в старом стандарте.

Контекст this никак не привязан к функции, даже если она создана в объявлении объекта. Чтобыthis передался, нужно вызвать функцию именно через точку (или квадратные скобки).

Любой более хитрый вызов приведёт к потере контекста, например:

|  |
| --- |
| name: "Вася",  hi: function() { alert(this.name); },  bye: function() { alert("Пока"); }  };  user.hi(); // Вася (простой вызов работает)  // а теперь вызовем user.hi или user.bye в зависимости от имени  (user.name == "Вася" ? user.hi : user.bye)(); // undefined |

В последней строке примера метод получен в результате выполнения тернарного оператора и тут же вызван. Но this при этом теряется.

Если хочется понять, почему, то причина кроется в деталях работы вызова obj.method().

Он ведь, на самом деле, состоит из двух независимых операций: точка . – получение свойства и скобки () – его вызов (предполагается, что это функция).

Функция, как мы говорили раньше, сама по себе не запоминает контекст. Чтобы «донести его» до скобок, JavaScript применяет «финт ушами» – точка возвращает не функцию, а значение специального «ссылочного» типа [Reference Type](https://people.mozilla.org/~jorendorff/es6-draft.html#sec-reference-specification-type).

Этот тип представляет собой связку «base-name-strict», где:

* *base* – как раз объект,
* *name* – имя свойства,
* *strict* – вспомогательный флаг для передачи use strict.

То есть, ссылочный тип (Reference Type) – это своеобразное «три-в-одном». Он существует исключительно для целей спецификации, мы его не видим, поскольку любой оператор тут же от него избавляется:

* Скобки () получают из base значение свойства name и вызывают в контексте base.
* Другие операторы получают из base значение свойства name и используют, а остальные компоненты игнорируют.

Поэтому любая операция над результатом операции получения свойства, кроме вызова, приводит к потере контекста.

Аналогично работает и получение свойства через квадратные скобки obj[method].

# Преобразование объектов

Бывают операции, при которых объект должен быть преобразован в примитив.

Например:

* Строковое преобразование – если объект выводится через alert(obj).
* Численное преобразование – при арифметических операциях, сравнении с примитивом.
* Логическое преобразование – при if(obj) и других логических операциях.

Рассмотрим эти преобразования по очереди.

Проще всего – с логическим преобразованием.

**Любой объект в логическом контексте – true, даже если это пустой массив [] или объект {}.**

|  |
| --- |
| if ({} && []) {  alert( "Все объекты - true!" ); // alert сработает  } |

Строковое преобразование проще всего увидеть, если вывести объект при помощи alert:

|  |
| --- |
| var user = {  firstName: 'Василий'  };  alert( user ); // [object Object] |

Как видно, содержимое объекта не вывелось. Это потому, что стандартным строковым представлением пользовательского объекта является строка "[object Object]".

Такой вывод объекта не содержит интересной информации. Поэтому имеет смысл его поменять на что-то более полезное.

**Если в объекте присутствует метод toString, который возвращает примитив, то он используется для преобразования.**

|  |
| --- |
| var user = {  firstName: 'Василий',  toString: function() {  return 'Пользователь ' + this.firstName;  }  };  alert( user ); // Пользователь Василий |

Все объекты, включая встроенные, имеют свои реализации метода toString, например:

|  |
| --- |
| alert( [1, 2] ); // toString для массивов выводит список элементов "1,2"  alert( new Date ); // toString для дат выводит дату в виде строки  alert( function() {} ); // toString для функции выводит её код |

Для численного преобразования объекта используется метод valueOf, а если его нет – то toString:

|  |
| --- |
| var room = {  number: 777,  valueOf: function() { return this.number; },  toString: function() { return this.number; }  };  alert( +room ); // 777, вызвался valueOf  delete room.valueOf; // valueOf удалён  alert( +room ); // 777, вызвался toString |

Метод valueOf обязан возвращать примитивное значение, иначе его результат будет проигнорирован. При этом – не обязательно числовое.

Итак, объект преобразован в примитив при помощи toString или valueOf.

Но на этом преобразования не обязательно заканчиваются. Вполне возможно, что в процессе вычислений этот примитив будет преобразован во что-то другое.

Например, рассмотрим применение к объекту операции ==:

|  |
| --- |
| var obj = {  valueOf: function() {  return 1;  }  };  alert( obj == true ); // true |

Объект obj был сначала преобразован в примитив, используя численное преобразование, получилось 1 == true.

Далее, так как значения всё ещё разных типов, применяются правила преобразования примитивов, результат: true.

То же самое – при сложении с объектом при помощи +:

|  |
| --- |
| var obj = {  valueOf: function() {  return 1;  }  };  alert( obj + "test" ); // 1test |

Или вот, для разности объектов:

|  |
| --- |
| var a = {  valueOf: function() {  return "1";  }  };  var b = {  valueOf: function() {  return "2";  }  };  alert( a + b ); // "12"  alert( a - b ); // "1" - "2" = -1 |

# Статические свойства и методы

Методы и свойства, которые не привязаны к конкретному экземпляру объекта, называют «статическими». Их записывают прямо в саму функцию-конструктор.

В коде ниже используются статические свойства Article.count и Article.DEFAULT\_FORMAT:

|  |
| --- |
| function Article() {  Article.count++;  }  Article.count = 0; // статическое свойство-переменная  Article.DEFAULT\_FORMAT = "html"; // статическое свойство-константа |

Они хранят данные, специфичные не для одного объекта, а для всех статей целиком.

Как правило, это чаще константы, такие как формат «по умолчанию» Article.DEFAULT\_FORMAT.

С примерами статических методов мы уже знакомы: это встроенные методы [String.fromCharCode](http://javascript.ru/String.fromCharCode),[Date.parse](http://javascript.ru/Date.parse).

Создадим для Article статический метод Article.showCount():

|  |
| --- |
| function Article() {  Article.count++;  //...  }  Article.count = 0;  Article.showCount = function() {  alert( this.count ); // (1)  }  // использование  new Article();  new Article();  Article.showCount(); // (2) |

Здесь Article.count – статическое свойство, а Article.showCount – статический метод.

Обратим внимание на использование this в примере выше. Несмотря на то, что переменная и метод – статические, он всё ещё полезен. В строке (1) он равен Article.

# Явное указание контекста

Итак, мы знаем, что this – это текущий объект при вызове «через точку» и новый объект при конструировании через new.

В этой главе наша цель получить окончательное и полное понимание this в JavaScript. Для этого не Синтаксис метода call: хватает всего одного элемента: способа явно указать this при помощи методов call и apply.

|  |
| --- |
| func.call(context, arg1, arg2, ...) |

При этом вызывается функция func, первый аргумент call становится её this, а остальные передаются «как есть».

**Вызов func.call(context, a, b...) – то же, что обычный вызов func(a, b...), но с явно указанным this(=context).**

Например, у нас есть функция showFullName, которая работает с this:

|  |
| --- |
| function showFullName() {  alert( this.firstName + " " + this.lastName );  } |

Пока объекта нет, но это нормально, ведь JavaScript позволяет использовать this везде. Любая функция может в своём коде упомянуть this, каким будет это значение – выяснится в момент запуска.

Вызов showFullName.call(user) запустит функцию, установив this = user, вот так:

|  |
| --- |
| function showFullName() {  alert( this.firstName + " " + this.lastName );  }  var user = {  firstName: "Василий",  lastName: "Петров"  };  // функция вызовется с this=user  showFullName.call(user) // "Василий Петров" |

После контекста в call можно передать аргументы для функции. Вот пример с более сложным вариантом showFullName, который конструирует ответ из указанных свойств объекта:

|  |
| --- |
| var user = {  firstName: "Василий",  surname: "Петров",  patronym: "Иванович"  };  function showFullName(firstPart, lastPart) {  alert( this[firstPart] + " " + this[lastPart] );  }  // f.call(контекст, аргумент1, аргумент2, ...)  showFullName.call(user, 'firstName', 'surname') // "Василий Петров"  showFullName.call(user, 'firstName', 'patronym') // "Василий Иванович" |

Если нам неизвестно, с каким количеством аргументов понадобится вызвать функцию, можно использовать более мощный метод: apply.

**Вызов функции при помощи func.apply работает аналогично func.call, но принимает массив аргументов вместо списка.**

|  |
| --- |
| func.call(context, arg1, arg2);  // идентичен вызову  func.apply(context, [arg1, arg2]); |

В частности, эти две строчки сработают одинаково:

|  |
| --- |
| showFullName.call(user, 'firstName', 'surname');  showFullName.apply(user, ['firstName', 'surname']); |

Преимущество apply перед call отчётливо видно, когда мы формируем массив аргументов динамически.

Например, в JavaScript есть встроенная функция Math.max(a, b, c...), которая возвращает максимальное значение из аргументов:

|  |
| --- |
| alert( Math.max(1, 5, 2) ); // 5 |

При помощи apply мы могли бы найти максимум в произвольном массиве, вот так:

|  |
| --- |
| var arr = [];  arr.push(1);  arr.push(5);  arr.push(2);  // получить максимум из элементов arr  alert( Math.max.apply(null, arr) ); // 5 |

В примере выше мы передали аргументы через массив – второй параметр apply… Но вы, наверное, заметили небольшую странность? В качестве контекста this был передан null.

Строго говоря, полным эквивалентом вызову Math.max(1,2,3) был бы вызов Math.max.apply(Math, [1,2,3]). В обоих этих вызовах контекстом будет объект Math.

Но в данном случае в качестве контекста можно передавать что угодно, поскольку в своей внутренней реализации метод Math.max не использует this. Действительно, зачем this, если нужно всего лишь выбрать максимальный из аргументов? Вот так, при помощи apply мы получили короткий и элегантный способ вычислить максимальное значение в массиве!

# Привязка контекста

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.

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, 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**

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

|  |
| --- |
| var f = user.sayHi;  setTimeout(f, 1000); // контекст user потерялиvar 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**

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**

В современном JavaScript (или при подключении библиотеки es5-shim для IE8-) у функций уже есть встроенный метод 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.

# Домашнее задание

* Объекты могут быть использованы для построения различных структур данных. Часто встречающаяся структура – список (не путайте с массивом). Список – связанный набор объектов, где первый объект содержит ссылку на второй, второй – на третий, и т.п.

|  |
| --- |
| var list = { value: 1, rest: { value: 2, rest: { value: 3, rest: null } } }; |

Списки удобны тем, что они могут делиться частью своей структуры. Например, можно сделать два списка, {value: 0, rest: list} и {value: -1, rest: list}, где list – это ссылка на ранее объявленную переменную. Это два независимых списка, при этом у них есть общая структура list, которая включает три последних элемента каждого из них. Кроме того, оригинальный список также сохраняет свои свойства как отдельный список из трёх элементов. Напишите функцию arrayToList, которая строит такую структуру, получая в качестве аргумента [1, 2, 3], а также функцию listToArray, которая создаёт массив из списка. Также напишите вспомогательную функцию prepend, которая получает элемент и создаёт новый список, где этот элемент добавлен спереди к первоначальному списку, и функцию nth, которая в качестве аргументов принимает список и число, а возвращает элемент на заданной позиции в списке, или же undefined в случае отсутствия такого элемента.

* Напишите две функции, reverseArray и reverseArrayInPlace. Первая получает массив как аргумент и выдаёт новый массив, с обратным порядком элементов. Вторая работает как оригинальный метод reverse – она меняет порядок элементов на обратный в том массиве, который был ей передан в качестве аргумента. Не используйте стандартный метод reverse.
* \* Реализовать функцию pick(obj, keys), которая принимает аргументами объект и массив строк (названия ключей). Возвращает новый объект, куда вошли все ключи, указанные в массиве keys, и соответствующие значения из объекта obj. Если в объекте obj, нет ключа, указанного в массиве keys, в результирующем объекте этот ключ не должен присутствовать.

|  |
| --- |
| var user = {  name: 'Sergey',  age: 30,  email: 'sergey@gmail.com',  friends: ['Sveta', 'Artem']  }  pick(user, ['name']); // {name: 'Sergey'}  pick(user, ['name', 'second-name']); // {name: 'Sergey'}  pick(user, ['name', 'friends']); // {name: 'Sergey', friends:['Sveta', 'Artem']} |

# Практика

Продолжаем разрабатывать адресную книгу. Практически во всех адресных книгах существует функция поиска контактов по имени и фамилии. Больше того, чтобы искать было удобно, поиск должен быть не только по полному совпадению, но и по частичному, а так же быть регистронезависмым. Причем неважно в каком порядке пользователь введет имя и фамилию, поиск все равно должен отработать корректно. Сегодня и мы реализуем такую функцию. Самый удобный способ поиска – использование регулярных выражений:

**function** **search**(value, users) {

**var** result = [];

**var** re = **new** RegExp(value, 'i');

users.forEach(**function**(**item**){

**var** findString = **item**.firstName + ' ' + **item**.lastName;

**if** (re.**test**(findString)) {

result.push(**item**);

}

})

**return** result;

}

Итоговый, получившийся файл, можно найти в материалах к уроку.

# Используемая литература

* [Mozilla Developer Network](https://developer.mozilla.org/)
* [MSDN](http://msdn.microsoft.com/)
* [Safari Developer Library](https://developer.apple.com/library/safari/navigation/index.html)
* [Современный учебник JavaScript](http://learn.javascript.ru)