

# Тонкости ECMA-262-3. Часть 3. This.



Read this article in: [English](#), [Chinese \(version1, version2, version 3\)](#), [Korean](#), [French](#).

1. Введение
2. Определение
3. This в глобальном коде
4. This в коде функции
  1. Тип Reference
  2. Вызов функции и НЕ тип Reference
  3. Тип Reference и значение this равное null
  4. This при вызове функции в качестве конструктора
  5. Явная установка значения this при вызове функций
5. Заключение
6. Дополнительная литература

## Введение

В данной небольшой заметке мы рассмотрим ещё одну сущность, связанную с контекстами исполнения. Речь пойдёт о ключевом слове `this`.

Как показывает практика, данная тема является достаточно сложной и часто вызывает затруднения в определении значения `this` в том или ином контексте.

Многие привыкли, что ключевое слово `this` в языках программирования тесно связано с объектно-ориентированным программированием, а именно, указывает на текущий порождаемый конструктором объект. В ECMAScript данная концепция также реализована, однако, как мы увидим, здесь `this` не ограничивается лишь определением порождаемого объекта.

Давайте подробнее разберём, что же такое `this` в ECMAScript.

## Определение

Итак, `this` является свойством контекста исполнения:

```
1 | activeExecutionContext = {  
2 |   VO: {...},  
3 |   this: thisValue  
4 | };
```

где VO — это **объект переменных** (variable object), который мы подробно разбирали в предыдущей статье.

Значение `this` напрямую связано с **типом исполняемого кода контекста**.

Определяется оно *при входе в контекст* и на протяжении исполнения кода контекста, является *неизменным*.

Рассмотрим эти случаи подробнее.

## This в глобальном коде

Здесь довольно всё просто. В коде глобального контекста, значением `this` всегда является сам *глобальный объект (global)*; таким образом, можно косвенно к нему обратиться:

```
1 // явное объявление свойства
2 // глобального объекта
3 this.a = 10; // global.a = 10
4 alert(a); // 10
5
6 // косвенное, посредством присваивания
7 // неопределённому до этого идентификатору
8 b = 20;
9 alert(this.b); // 20
10
11 // также косвенное, посредством объявления
12 // переменной, поскольку объектом переменных
13 // в глобальном контексте является сам глобальный объект
14 var c = 30;
15 alert(this.c); // 30
```

## This в коде функции

Со значением `this` в коде функции — интереснее. Именно в этом моменте возникает большинство затруднений.

Первая (и, возможно, главная) особенность значения `this` в этом типе кода заключается в том, что оно здесь *не связано статично* с функцией.

Как уже было сказано выше, значение `this` определяется при входе в контекст, и в случае с кодом функции, *каждый раз может быть абсолютно разным*.

Однако, на протяжении исполнения кода контекста, значение `this` является *неизменным*, т.е. нельзя присвоить ему новое значение динамически в рантайме, т.к. `this` *не является переменной* (в отличие, скажем, от языка программирования *Python*, и его явно определяемого объекта `self`, значение которого может неоднократно меняться по ходу исполнения кода контекста):

```
1 var foo = {x: 10};
2
3 var bar = {
4   x: 20,
5   test: function () {
6
7     alert(this === bar); // true
8     alert(this.x); // 20
9
10    this = foo; // ошибка, нельзя менять this
11
12    alert(this.x); // если бы не было ошибки, было бы 10, а не 20
13
14   }
15 };
16
17 // при входе в контекст, значение this
18 // определено как объект "bar"; почему – будет
19 // подробно разобрано ниже
20
21 bar.test(); // true, 20
22
23 foo.test = bar.test;
24
25 // однако, здесь уже this указывает
26 // на "foo" - при вызове той же функции
27
28 foo.test(); // false, 10
```

Итак, от чего же зависит меняющееся значение `this` в коде функции? Здесь есть несколько факторов.

Во-первых, при обычном вызове функции, `this` определяется *вызывающей стороной*, которая активирует код контекста функции, — так называемый, *caller*, т.е. родительский контекст, который вызывает функцию. Определение значения `this` происходит по *форме выражения вызова* (иными словами, как *синтаксически* вызвана функция).

Это очень важный момент, его нужно понять, запомнить и чётко осознавать, тогда, с определением значения `this` в любом контексте, никаких проблем возникать не будет. Именно *форма выражения вызова* влияет на значение `this` вызываемого контекста, и *ничто другое*.

Так, например, в некоторых статьях и даже книгах по JavaScript, иногда можно видеть неверные утверждения, где описания вроде: “`this` зависит от того, как описана функция: если это глобальная функция, то `this` будет указывать на глобальный объект, если же функция является методом объекта, то `this` всегда будет объектом, которому принадлежит вызываемый метод” — являются *ошибочными*. Забегая вперёд, покажем, что уже обычную глобальную функцию можно активировать *разными формами вызова*, которые и влияют на разное значение `this`:

```
1 function foo() {
2   alert(this);
3 }
4
5 foo(); // global
6
7 alert(foo === foo.prototype.constructor); // true
8
9 // но при иной форме вызова той же
10 // функции, this будет иметь уже другое значение
11
12 foo.prototype.constructor(); // foo.prototype
```

Аналогично можно вызвать функцию, описанную как метод объекта, но значением `this` не будет являться этот объект:

```
1 var foo = {
2   bar: function () {
3     alert(this);
4     alert(this === foo);
5   }
6 };
7
8 foo.bar(); // foo, true
9
10 var exampleFunc = foo.bar;
11
12 alert(exampleFunc === foo.bar); // true
13
14 // опять же, при иной форме вызова той же
15 // функции, this уже установлен в другое значение
16
17 exampleFunc(); // global, false
```

Каким же образом форма выражения вызова влияет на значение `this`? Здесь всё завязано на один из внутренних типов реализации — тип `Reference`, который, для полного понимания определения значения `this`, нужно рассмотреть подробней.

## Тип Reference

Псевдокодом, значение типа `Reference` можно представить в виде обычного объекта с двумя свойствами: *база (base)* (т.е. объект, которому принадлежит

свойство) и имя свойства (*property name*) внутри базы:

```
1 | var valueOfReferenceType = {
2 |   base: <base object>,
3 |   propertyName: <property name>
4 | };
```

Значение типа `Reference` может быть получено *только в двух случаях*:

1. когда мы имеем дело с *идентификатором* (*identifier*);
2. либо же с *выражением доступа к свойству* (*property accessor*);

Процесс *разрешения имён идентификаторов* (*identifier resolution*) подробно рассматривается в *четвёртой части* (Цепь областей видимости, Scope chain). Здесь же отметим, что на выходе данного алгоритма *всегда* будет значение типа `Reference` (это важно для значения `this`).

Идентификаторами являются имена переменных, функций, формальных параметров функций и неявные свойства глобального объекта. К примеру, для значений по следующим идентификаторам:

```
1 | var foo = 10;
2 | function bar() {}
```

в промежуточных операциях, будут получены соответствующие значения типа `Reference`:

```
1 | var fooReference = {
2 |   base: global,
3 |   propertyName: 'foo'
4 | };
5 |
6 | var barReference = {
7 |   base: global
8 |   propertyName: 'bar'
9 | };
```

Для получения *реального* значения объекта из значения типа `Reference` предусмотрен метод `GetValue`, который псевдокодом можно описать следующим образом:

```
1 | function GetValue(value) {
2 |
3 |   if (Type(value) !== Reference) {
4 |     return value;
5 |   }
6 |
7 |   var base = GetBase(value);
8 |
9 |   if (base === null) {
10 |    throw new ReferenceError;
11 |   }
12 |
13 |   return base.[[Get]](GetPropertyname(value));
14 |
15 | }
```

где внутренний метод `[[Get]]` получает значение свойства объекта, учитывая также и наследуемые свойства из цепи прототипов:

```
1 | GetValue(fooReference); // 10
2 | GetValue(barReference); // function object "bar"
```

Выражение *доступа к свойству* (*property accessor*), так же многим известно. Осуществляется оно либо через *точечную нотацию* (когда имя свойства является правильным идентификатором и заранее известно), либо же — через *скобочную*:

```
1 | foo.bar();
2 | foo['bar']();
```

На выходе промежуточного вычисления также будет получено значение типа

Reference :

```
1 | var fooBarReference = {
2 |   base: foo,
3 |   propertyName: 'bar'
4 | };
5 |
6 | GetValue(fooBarReference); // function object "bar"
```

Итак, как же связано значение типа `Reference` со значением `this` контекста функции? — *Самым главным образом*. Данный момент является основным в этой статье. Правило определения `this` в контексте функции звучит следующим образом:

Значение `this` в контексте функции определяется *вызывающей стороной (caller-ом) по форме вызова*.

Если слева от скобок вызова ( `...` ), находится выражение типа `Reference`, то значением `this` будет являться *базовый объект* этого значения типа `Reference`.

Во всех остальных случаях (т.е. при любом другом типе значения, отличном от типа `Reference`), значением `this` будет *всегда* являться `null`. Но, т.к. `null` особого смысла для значения `this` не несёт, автоматом подставляется *глобальный объект*.

Покажем на примерах:

```
1 | function foo() {
2 |   return this;
3 | }
4 |
5 | foo(); // global
```

Видим, что слева от скобок вызова стоит выражение типа `Reference` (поскольку `foo` — это идентификатор):

```
1 | var fooReference = {
2 |   base: global,
3 |   propertyName: 'foo'
4 | };
```

Соответственно, значение `this` определено, как база этого значения типа `Reference`, т.е. глобальный объект. Аналогично с выражением доступа к свойству:

```
1 | var foo = {
2 |   bar: function () {
3 |     return this;
4 |   }
5 | };
6 |
7 | foo.bar(); // foo
```

Имеем, опять же, значение типа `Reference`, где базой является объект `foo`, который и будет использован в качестве значения `this` при активации функции `bar`:

```

1 | var fooBarReference = {
2 |   base: foo,
3 |   propertyName: 'bar'
4 | };

```

Однако, активируя *ту же самую функцию*, но с *другой формой вызова*, мы имеем уже другое значение `this`:

```

1 | var test = foo.bar;
2 | test(); // global

```

поскольку `test`, являясь идентификатором, порождает *другое значение типа* `Reference`, то именно база этого *нового* значения типа `Reference` и будет использована в качестве значения `this` — т.е. *глобальный объект*:

```

1 | var testReference = {
2 |   base: global,
3 |   propertyName: 'test'
4 | };

```



Примечание: в *строгом режиме ES5* значение `this` не преобразуется к глобальному объекту, но установлено вместо этого в значение `undefined`.

Теперь мы можем точно сказать, почему одна и та же функция, но активированная *разными формами вызова*, имеет и разные значения `this` — ответ заключён в разных промежуточных значениях типа `Reference`:

```

1 | function foo() {
2 |   alert(this);
3 | }
4 |
5 | foo(); // global, т.к.
6 |
7 | var fooReference = {
8 |   base: global,
9 |   propertyName: 'foo'
10 | };
11 |
12 | alert(foo === foo.prototype.constructor); // true
13 | // другая форма вызова
14 |
15 | foo.prototype.constructor(); // foo.prototype, т.к.
16 |
17 | var fooPrototypeConstructorReference = {
18 |   base: foo.prototype,
19 |   propertyName: 'constructor'
20 | };
21 |

```

Ещё (классический) пример динамического определения `this` по форме выражения вызова:

```

1 | function foo() {
2 |   alert(this.bar);
3 | }
4 |
5 | var x = {bar: 10};
6 | var y = {bar: 20};
7 |
8 | x.test = foo;
9 | y.test = foo;
10 |
11 | x.test(); // 10
12 | y.test(); // 20

```

## Вызов функции и НЕ тип Reference

Итак, как мы уже отметили, в случае, когда слева от скобок вызова находится значение *не* типа `Reference`, а *любого другого* типа, значение `this` будет автоматически определено как `null`, и, как следствие, *global*.

Рассмотрим примеры таких выражений:

```
1 | (function () {  
2 |   alert(this); // null => global  
3 | })();
```

В данном случае мы имеем объект `Function`, но не объект типа `Reference` (это не идентификатор и не выражение доступа к свойству), соответственно, значение `this` в конечном итоге будет определено, как глобальный объект.

Примеры сложнее:

```
1 | var foo = {  
2 |   bar: function () {  
3 |     alert(this);  
4 |   }  
5 | };  
6 |  
7 | foo.bar(); // Reference, OK => foo  
8 | (foo.bar)(); // Reference, OK => foo  
9 |  
10 | (foo.bar = foo.bar)(); // global?  
11 | (false || foo.bar)(); // global?  
12 | (foo.bar, foo.bar)(); // global?
```

Почему же, используя выражение доступа к свойству (*property accessor*), промежуточным результатом которого должно являться значение типа `Reference`, мы, в определённых вызовах, получаем в качестве `this` не базовый объект (т.е. `foo`), а `global`?

Дело в том, что последние три вызова, после применения определённых операций, имеют слева от скобок вызова уже значение не типа `Reference`.

С первым случаем всё понятно — там однозначно тип `Reference` и, как следствие, `this` — это база, т.е. `foo`.

Во втором случае применяется оператор группировки, который не вызывает, рассмотренный выше, метод получения реального значения объекта из значения типа `Reference`, т.е. `GetValue` (см. примечание к 11.1.6). Соответственно, на выходе оператора группировки — всё ещё значение типа `Reference`, а потому, значение `this` снова определено, как база, т.е. `foo`.

В третьем случае, оператор присваивания, в отличие от оператора группировки, вызывает метод `GetValue` (см. шаг 3 11.13.1). В итоге на выходе уже будет значения типа `Function`, означающее, что в качестве `this` будет использован `null` и, как следствие, `global`.

Аналогично с четвертым и пятым случаями — оператор запятая и логическое ИЛИ вызывают `GetValue`, соответственно, мы теряем значение типа `Reference` и получаем значение типа `Function`; вновь, `this` определён как `global`.

## Тип `Reference` и значение `this` равно `null`

Существует ситуация, когда выражение вызова определит слева от скобок вызова значение типа `Reference`, однако значение `this` будет определено как `null` и, как следствие, `global`.

Это относится к случаю, когда базовым объектом значения типа `Reference`, является объект активации.

Данную ситуацию можно показать на примере с вложенной функцией, вызванной из родительской. Как нам известно из второй части, локальные переменные, локальные функции и параметры функции хранятся в *объекте активации* данной функции:

```
1 function foo() {
2   function bar() {
3     alert(this); // global
4   }
5   bar(); // равносильно A0.bar()
6 }
```

Объект активации всегда возвращает в качестве значения `this` — `null` (т.е. схематичная запись `A0.b()` равносильна `null.b()`). И здесь мы снова возвращаемся к вышеописанному случаю, и снова в качестве `this` подставляется *глобальный объект*.

Исключение может составить вызов внутренней функции в блоке оператора `with`, в случае, если объект `with` содержит свойство с именем функции. Оператор `with` в *цепи областей видимости* добавляет свой объект *перед* объектом активации. Соответственно, имея значения типа `Reference` (по идентификатору или выражению доступа к свойству), мы имеем базой *не объект активации*, а объект `with`. Кстати, это касается не только вложенной функции, но и глобальной — объект `with` “заслонит” вышестоящий объект (глобальный или объект активации) области видимости:

```
1 var x = 10;
2
3 with ({
4   foo: function () {
5     alert(this.x);
6   },
7   x: 20
8 }) {
9
10  foo(); // 20
11
12 }
13
14 // поскольку
15
16 var fooReference = {
17   base: __withObject,
18   propertyName: 'foo'
19 };
20
21 
```

Аналогичная ситуация должна быть с активацией функции, являющейся параметром выражения `catch`: в данном случае объект `catch` также добавляется *перед* объектом активации, либо глобальным объектом. Однако, данное поведение было определено как баг ЕСМА-262-3 и исправлено в новой версии стандарта ЕСМА-262-5; таким образом, значение `this` в данной активации должно быть `global`, но не объект `catch`:

```
1 try {
2   throw function () {
3     alert(this);
4   };
5 } catch (e) {
6   e(); // __catchObject - в ES3, global - исправлено в ES5
7 }
8
9 // по идее
10
11 var eReference = {
12   base: __catchObject,
13   propertyName: 'e'
14 };
15
16 // однако, это баг, и base
17 // принудительно определяется как
18 // null => global
19
20 var eReference = {
```



```
21   base: global,  
22   propertyName: 'e'  
23 };
```

Та же самая ситуация с рекурсивным вызовом **именованной функции-выражения** (подробней о функциях смотрите в [пятой части](#)). При первой активации функции, базой является родительский объект активации (или глобальный объект), при последующих — базой должен быть специальный объект хранящий имя функции-выражения. Однако, в данном случае в качестве `this` также всегда используется `global`:

```
1  (function foo(bar) {  
2  
3    alert(this);  
4  
5    !bar && foo(1); // "должен" быть спец.объект, но всегда global  
6  
7  })(); // global
```

## This при вызове функции в качестве конструктора

Есть ещё одна ситуация, связанная со значением `this` в контексте функции — это вызов функции в качестве конструктора:

```
1  function A() {  
2    alert(this); // вновь созданный объект, ниже - объект "a"  
3    this.x = 10;  
4  }  
5  
6  var a = new A();  
7  alert(a.x); // 10
```

В данном случае, оператор `new` вызовет внутренний метод `[[Construct]]` функции `A`, который, в свою очередь, после создания объекта, вызовет внутренний метод `[[Call]]`, всё той же функции `A`, передав в качестве значения `this` вновь созданный объект.

## Явная установка значения this при вызове функций

Существуют два метода, описанные в `Function.prototype` (а, соответственно, они доступны всем функциям), позволяющие явно указать значение `this` при вызове функции. Это методы `apply` и `call`.

Оба они принимают в качестве первого параметра значение `this`, которое будет использовано в контексте вызова. Разница между этими методами незначительная: для первого из них вторым параметром обязательно должен быть массив (либо *массиво-подобный объект*, например, `arguments`), в свою очередь, `call` может принимать любые параметры. Обязательным параметром для обоих методов является лишь первый — значение `this`.

Примеры:

```
1  var b = 10;  
2  
3  function a(c) {  
4    alert(this.b);  
5    alert(c);  
6  }  
7  
8  a(20); // this === Global, this.b == 10, c == 20  
9  
10 a.call({b: 20}, 30); // this === {b: 20}, this.b == 20, c == 30  
11 a.apply({b: 30}, [40]) // this === {b: 30}, this.b == 30, c == 40
```

# Заключение

В данной заметке мы разобрали особенности ключевого слова `this` в ECMAScript (и они, действительно — *особенности*, в отличие, скажем, от C++ или Java). Надеюсь, статья помогла более чётко представить, как работает эта сущность в JavaScript. Как всегда, буду рад ответить на ваши вопросы в комментариях.

## Дополнительная литература

10.1.7 – This;

11.1.1 – Ключевое слово `this`;

11.2.2 – Оператор `new`;

11.2.3 – Вызовы функций.

**Автор:** Dmitry A. Soshnikov

**Дополнения и корректировки:** Zeroglif

**Дата публикации:** 29.06.2009; **обновление:** 07.03.2010;

### Archives

March 2020

October 2019

August 2019

July 2019

February 2019

December 2017

November 2017

October 2016

September 2016

February 2016

January 2016

September 2015

September 2014

August 2014

July 2011

February 2011

January 2011

December 2010

September 2010

June 2010

April 2010

March 2010

February 2010

November 2009

September 2009

July 2009

June 2009

## Meta


[Log in](#)



**Dmitry Soshnikov**

*Software engineer interested in learning and education.  
Sometimes blog on topics of programming languages theory,  
compilers, and ECMAScript.*

**Published**  
2009-06-29

 [Write a Comment](#)

 20 COMMENTS



**Alex**

2011-09-14

Большое спасибо за статью, очень актуальный вопрос.



**Sergey**

2012-01-30

Дмитрий спасибо за замечательную статью!

Объясните пожалуйста чем отличается `(foo.bar = foo.bar)()`  
от `foo.bar = foo.bar; foo.bar();`

И ещё вопрос о том же, я знаю что чтобы вызвать `eval` (да и в принципе любую функцию) в глобальном контексте можно написать что то вроде `(eval=eval)("/ *global code*/");` или `(1,eval)("");`  
Такая техника называется indirect call.

Могли бы вы пояснить логику выполнения этого выражения?

Спасибо.



**Dmitry Soshnikov**

@Sergey

```
чем отличается (foo.bar = foo.bar)()
от foo.bar = foo.bar; foo.bar();
```

В первом случае, как было отмечено в этой статье, оператор присваивания вызывает внутренний метод `GetValue`. Данный метод в свою очередь “портит” значение типа `Reference`, получая истинное значение (саму функцию `foo.bar`).

Потеря значения типа `Reference` приводит к глобальному `this` в ES3 (в ES5 в `strict-mode` будет `undefined` в качестве `this`).

Обратите еще раз внимание, что *просто* оператор группировки (обрамляющие скобки), *не вызывает* `GetValue`, и таким образом вызов функции получает `this` как объект `foo : (foo.bar)();`.

Во втором же случае у Вас просто два независимых действия. Первое никак не относится к определению `this`; Вы просто “переприсвоили” функции значение самой себя. А вот дальше уже идет активация функции, и именно в этот момент определяется `this`, как объект `foo`, поскольку слева от скобок вызова — `Reference`, т.е. *аксессор* — доступ к свойству через точку.

```
(eval=eval)("/global code*"); или (1,eval)("");
```

Такая техника называется *indirect call*.

Да, есть такой вызов `eval 'a'`. Подробней я его описывал в статье про `strict-mode`.

Суть его опять же сводится к `Reference` типу. Проще — только `eval` записанный в данной синтаксической форме, является *явным* (*direct*):

```
1 | eval(...);
```

Все остальные синтаксические формы — это уже *косвенные* (*indirect*) вызовы и исполняются в глобальном контексте. Причины этого связаны с особенностями реализации `eval 'a'` и с безопасностью.



Aleksey

2012-02-13

Дмитрий, у Вас отличные и **главное — интересные** статьи! Спасибо за это! Объясните, пожалуйста, подробнее, какое же все-таки значение будет принимать ключевое слово `this` в тот момент, когда мы только входим в контекст исполнения. Предположим, у нас есть только глобальный контекст исполнения программы. У него определяется свойство `this`. Какое значение оно будет принимать до интерпретации программы? В Вашем примере использовалось `thisValue`. Что оно представляет из себя? Заранее огромное спасибо!



Dmitry Soshnikov

2012-02-13

@Aleksey

Объясните, пожалуйста, подробнее, какое же все-таки значение будет

принимать ключевое слово `this` в тот момент, когда мы только входим в контекст исполнения.

Значение `this` всегда определяется до интерпретации программы (т.е., когда мы видим какой-то код и начинаем его исполнять — уже точно известно, что содержит `this`, и это значение неизменно на протяжении исполнения кода контекста).

Предположим, у нас есть только глобальный контекст исполнения программы. У него определяется свойство `this`. Какое значение оно будет принимать до интерпретации программы?

Да, верно, и это значение — сам глобальный объект, как было отмечено выше.

В Вашем примере использовалось `thisValue`. Что оно представляет из себя?

Это схематическое обозначение значения `this` в определенном контексте исполнения.

Структура контекста следующая (всего три основных свойства):

```
1 | executionContext = {
2 |   VO: { хранилище_переменных }, // variable object
3 |   ScopeChain: VO + все родительские VO,
4 |   this: значение this // thisValue
5 | };
```

Если есть необходимость, я рекомендую начать с описания самого контекста исполнения, и дальше уже его компонентов: `VO`, `ScopeChain` и `this`. Или же прочитать обзорную лекцию [JavaScript. Ядро](#), где все эти моменты так же затрагиваются.



**Aleksey**

2012-02-16

Дмитрий, спасибо за Ваши пояснения! Продолжаю дальше читать Ваши статьи.



**max**

2012-05-10

в примере “Вызов функции и НЕ тип Reference” команда `(false || foo.bar)();` возвращает `true`



**maksimr**

2012-05-13

@max

Если вы говорите про этот пример:

```
1 | var foo = {
2 |   bar: function () {
3 |     alert(this);
4 |   }
5 | };
```

```
6 | (false || foo.bar());
```

То это выражение возвращает `undefined`. Тут основной смысл заключается в том какое значение присвоено ключевому слову `this`, и в данном, конкретном, случае `this` ссылается на глобальный объект.

Надеюсь я правильно понял ваш вопрос, и смог на него ответить.



**Иван**

2012-11-15

Добрый день!

В FireFox (версии 16.0.2) код

```
1 | var f = {
2 |   b: function() {
3 |     console.log(this);
4 |   }
5 | };
6 | f.b();
```

пишет в консоль

```
1 | Object { b=function() }
```

У вас же в примере указано, что будет `"f"`:

```
1 | var foo = {
2 |   bar: function () {
3 |     alert(this);
4 |   }
5 | };
6 | foo.bar(); // Reference, OK => foo
```



**Jmunb**

2012-11-24

@Иван Иван

Я думаю, дело в том, что при использовании консоли браузера, код будет выполнен с помощью функции `eval`, это и приведет к изменению значения `this`.



**A.i.**

2012-12-14

А что Вам консоль должна выдать?

Все правильно, Вы получили свой объект `f`, с методом `b`.

В случае, если `!Reference`, ответом был бы глобальный объект, т.к. это браузерная консоль, то Вы бы получили объект `Window`



**Konstantin**

2012-12-24

Почему в данном коде `this` – global?

```
1 | var someObj = {
2 |   someProp : function() {
3 |     outerFunc();
4 |   };
5 |   function outerFunc() { alert(this); }
6 |
7 |   someObj.someProp();
```



**Alexander**

2013-08-09

Дмитрий подскажите пожалуйста, я правильно мыслю?

```
1 | 'use strict';
2 |
3 | var test = {
4 |   aa : function () {
5 |     var bb = {
6 |       j : function () {
7 |         console.log(this);
8 |       }
9 |     };
10 |
11 |     function bar () {
12 |       return bb.j;
13 |     }
14 |
15 |     return bar();
16 |   }
17 | };
18 |
19 | test.aa(); // this === undefined;
```

```
1 | test.aa === Reference // потому, что "."
2 | test.aa -> bar === Reference // потому, что FD
3 | test.aa -> bar -> bb.j === Reference // потому, что "."
4 | var jReference = {
5 |   base: bb,
6 |   propertyName: 'j'
7 | };
```

НО при выполнении `bar()` вернётся ссылка на функцию jFunc `j : func...`

Соответственно, когда:

```
1 | jFunc()
```

мы попадаем под правило

| *HE mun Reference*

т.к.

```
1 | jFunc() !== identifier
```

и

```
1 | jFunc -> console.log(this); // undefined т.к. 'use strict';
```

Всё верно ?



**Alexander**

2013-08-10

*Konstantin :*

*Почему в данном коде `this` – `global`?*

Если объяснять чуть проще, главное то, “как” мы вызываем функцию, в которой используем `this`.

То есть не “откуда”, а именно “как”!

```
outerFunc(); -> Вызывается из АО родительской somePropFE, а доступа к АО в нормальных браузерах (текущий костяк) нет. Соответственно this === AOsomePropFunctionExpression === global // или undefined в 'use strict';
```

Можно ещё так сказать:

```
[ТО ЧТО СТОИТ ЗДЕСЬ, ВЛИЯЕТ НА this ВНУТРИ ->] outerFunc [ЕСЛИ ДАЛЬШЕ ИДЁТ ВЫЗОВ ->]
()
```



**Dmitry Soshnikov**

2013-08-17

**@Alexander**

*НО при выполнении `bar()` вернётся ссылка на функцию `jFunc`*

Пример получился несколько запутанный, но в целом, да, все верно 😊



**Konstantin**

2013-11-03

**@Alexander**

Не очень понял Ваш ответ по поводу моего примера выше.

Вообщем, как я вижу ситуацию:

Ф-ция `outerFunc` создаётся в глобальном контексте, то есть входит в АО глобального объекта. В теле свойства `someProp` она вызывается как обычная ф-ия, следовательно её промежуточный Reference объект в качестве базы имеет `global`, обращение к этой функции идёт через `scope chain`. Поэтому и получаем `global` в качестве `this`...

Надеюсь на Ваш ответ 😊



**Кирилл**

2015-09-24

Поясните, пожалуйста.

*Во-первых, при обычном вызове функции, `this` определяется вызывающей стороной, которая активирует код контекста функции, — так называемый, *caller*, т.е. родительский контекст, который вызывает функцию.*



И вот:

Если слева от скобок вызова ( ... ), находится выражение типа *Reference*, то значением `this` будет являться базовый объект этого значения типа *Reference*.

Однако ниже говорится, что:

Существует ситуация, когда выражение вызова определит слева от скобок вызова значение типа *Reference*, однако значение `this` будет определено как `null` и, как следствие, *global*.

Это относится к случаю, когда базовым объектом значения типа *Reference*, является объект активации.

Следовательно немного не сходится пример:

```
1 function foo() {
2   function bar() {
3     alert(this); // global
4   }
5   bar(); // равносильно AO.bar()
6 }
```

Как *base* может быть АО(объектом активации), если в определении сказано, что *base* – родительский контекст исполнения, а VO/АО – является именно СВОЙСТВОМ контекста исполнения. (Взято из второго урока про контексты исполнения).

Понятно, что пример раскладывается так:

```
1 var fooReference = {
2   base: global,
3   propertyName: 'foo'
4 };
5 var barReference = {
6   base: AO,
7   propertyName: 'bar'
8 };
```

Однако следуя логике определений про то, что *base* является именно родительским контекстом исполнения (как я понял, не свойство родительского контекста исполнения VO/АО, а им самим), то должно быть так:

```
1 var fooReference = {
2   base: global,
3   propertyName: 'foo'
4 };
5 var barReference = {
6   base: foo functionContext,
7   propertyName: 'bar'
8 };
```



Dmitry Soshnikov

2015-09-28

@Кирилл

Вероятно я несколько путано написал первое предложение. Сам контекст исполнения *никогда не является* значением свойства `base` объекта типа *Reference*. Соответственно, правильный вариант — первый:

```
1 var fooReference = {
2   base: global,
3   propertyName: 'foo'
4 };
```

```

4   };
5   var barReference = {
6     base: AO,
7     propertyName: 'bar'
8   };

```

Если перефразировать проще первое предложение, то в нем говорится, что значение `this` не зависит от того, как функция была создана, а от того, как функция (и где) будет *вызвана*, т.е. от caller'a:

```

1   var foo = {
2     x: 10,
3     bar: function() {
4       return this.x;
5     },
6   };
7
8   var bar = foo.bar;
9
10  var x = 100;
11
12  bar(); // 100, global.bar();
13
14  function baz() {
15    bar(); // 100, AO.bar() -> null.bar() -> global.bar();
16  }
17
18  bar();

```

Видим, что функция создана как метод объекта `foo`, но на определение значения `this` это никак не повлияло, поскольку caller (глобальный контекст) определил его при вызове — передав глобальный объект в качестве `this` (Reference: `{base: global, propertyName: 'bar'}`). В случае вызова из внутренней функции, base будет объект активации, и дальше также `global`.



**Dmitriy**

2016-03-14

```

1   var user = {
2     firstName: "Василий",
3
4     export: this
5   };
6
7   alert( user.export.firstName );

```

не подскажите а почему алерт выдает `undefined`, а вот здесь ожидаемо `var name = ""`;

```

1   var user = {
2     name: "Василий",
3
4     export: function() {
5       return {
6         value: this
7       };
8     }
9   };
10
11
12  alert( user.export().value.name );

```




**Dmitry Soshnikov**

2016-03-14

@Dmitriy

| не подскажите а почему алерт выдает `undefined`

Потому что контекст, в котором *создается* объект не равен самому объекту. Только внутри функции `this` будет объектом, но не в момент создания объекта. В момент создания `this` равен глобальному объекту, у которого нет свойства `firstName`.

 Write a Comment

RELATED CONTENT BY TAG [ECMA-262-3](#) [ECMAScript](#) [Russian](#) [this](#)