О ключевом слове «this» языка JavaScript: особенности использования с пояснениями

* [Переводы](https://tproger.ru/category/translations/), 23 августа 2016 в 23:32
* 42 062



Перевод статьи [«Gentle explanation of 'this' keyword in JavaScript»](https://rainsoft.io/gentle-explanation-of-this-in-javascript/)

*Рассказывает*[*Дмитрий Павлютин*](https://rainsoft.io/author/dmitri-pavlutin/)

Тайна this

Долгое время ключевое слово this оставалось для меня загадкой. Это мощный инструмент, но разобраться в нём нелегко.

С точки зрения Java, PHP или любого другого обычного языка [this](https://en.wikipedia.org/wiki/This_(computer_programming)" \t "_blank) расценивается как экземпляр текущего объекта в методе класса, не больше и не меньше. Чаще всего его нельзя использовать вне метода, и этот подход не вызывает непонимания.

В JavaScript this — это текущий контекст исполнения функции. Поскольку функцию можно вызвать четырьмя способами:

* вызов функции: alert('Hello World!'),
* вызов метода: console.log('Hello World!'),
* вызов конструктора: new RegExp('\\d'),
* непрямой вызов: alert.call(undefined, 'Hello World!'),

и каждый из них определяет свой контекст, поведение this слегка не соответствует ожиданиям начинающих разработчиков. Кроме того, [strict mode](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Strict_mode" \t "_blank) также влияет на контекст исполнения.

Ключом к пониманию ключевого слова this является осознание принципов вызова функции и его влияния на контекст. В этой статье рассказывается про вызовы функций, влияние вызовов на this и типичные ловушки при идентификации контекста.

**[Конкурс приложений VK Mini Apps](https://tproger.ru/events/vk-mini-apps-contest/?utm_source=in_text)**

[26 ноября – 26 декабря, онлайн, беcплатно](https://tproger.ru/events/vk-mini-apps-contest/?utm_source=in_text)

[tproger.ru](https://tproger.ru/events/vk-mini-apps-contest/?utm_source=in_text)

[События и курсы на tproger.ru](https://tproger.ru/events/)

Прежде чем мы начнём, давайте познакомимся с несколькими терминами:

* **Вызов** — это исполнение кода тела функции. Например, вызовом функции parseInt будет parseInt('15').
* **Контекстом** вызова является значение this в теле функции.
* **Область видимости функции** — это набор переменных, объектов и функций, к которым можно получить доступ из тела функции.

Вызов функции

**Вызов функции** совершается, когда за выражением, являющимся объектом функции, следуют открывающая скобка (, разделённый запятыми список аргументов и закрывающая скобка ), например, parseInt('18'). Выражение не может быть [аксессором](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Property_accessors" \t "_blank) myObject.myFunction, который совершает вызов метода. Например, [1,5].join(',') — это вызов не функции, а метода.

Простой пример вызова функции:

function hello(name) {

return 'Hello ' + name + '!';

}

// Function invocation

var message = hello('World');

console.log(message); // => 'Hello World!'

hello('World') — это вызов функции: hello расценивается как объект функции, за которым в скобках следует аргумент 'World'.

Более сложный пример — [немедленно вызываемая функция](https://en.wikipedia.org/wiki/Immediately-invoked_function_expression):

var message = (function(name) {

return 'Hello ' + name + '!';

})('World');

console.log(message); // => 'Hello World!'

Это тоже вызов функции: первая пара скобок (function(name) {...}) расценивается как объект функции, за которым в скобках следует аргумент: ('World').

this при вызове функции

this — это глобальный объект при вызове функции

Глобальный объект определяется средой исполнения. В веб-браузере это объект [window](https://developer.mozilla.org/en-US/docs/Web/API/Window" \t "_blank).

В вызове функции контекстом исполнения является глобальный объект. Давайте проверим контекст следующей функции:

function sum(a, b) {

console.log(this === window); // => true

this.myNumber = 20; // add 'myNumber' property to global object

return a + b;

}

// sum() is invoked as a function

// this in sum() is a global object (window)

console.log(sum(15, 16)); // => 31

console.log(window.myNumber); // => 20

Когда вызывается sum(15, 16), JavaScript автоматически инициализирует this как глобальный объект, являющийся window в браузере.

Когда this используется вне области видимости какой-либо функции (самая внешняя область видимости: контекст глобального исполнения), он также относится к глобальному объекту:

console.log(this === window); // => true

this.myString = 'Hello World!';

console.log(window.myString); // => 'Hello World!'

<script type="text/javascript">

console.log(this === window); // => true

</script>

this при вызове функции в strict mode

this принимает значение undefined при вызове функции в strict mode

Strict mode был введён в [ECMAScript 5.1](https://www.ecma-international.org/ecma-262/5.1/" \l "sec-10.1.1" \t "_blank) и представляет собой более надёжную систему защиты и проверки ошибок. Для активации поместите директиву 'use strict' вверху тела функции. Этот режим влияет на контекст исполнения, заставляя this быть undefined. Контекст исполнения перестаёт быть глобальным объектом, в отличие от предыдущего случая.

Пример функции, запущенной в strict mode:

function multiply(a, b) {

'use strict'; // enable the strict mode

console.log(this === undefined); // => true

return a \* b;

}

// multiply() function invocation with strict mode enabled

// this in multiply() is undefined

console.log(multiply(2, 5));

Когда multiply(2, 5) вызывается this становится undefined.

Strict mode активен не только в текущей области видимости, но и во всех вложенных:

function execute() {

'use strict'; // activate the strict mode

function concat(str1, str2) {

// the strict mode is enabled too

console.log(this === undefined); // => true

return str1 + str2;

}

// concat() is invoked as a function in strict mode

// this in concat() is undefined

console.log(concat('Hello', ' World!'));

}

execute();

'use strict' вставлена вверху тела execute, что активирует strict mode внутри её области видимости. Поскольку concat объявлена внутри области видимости execute, она наследует strict mode. И вызов concat('Hello', ' World!') приводит к тому, что this становится undefined.

Один файл JavaScript может содержать как «строгие», так и «нестрогие» функции. Поэтому возможно иметь в одном скрипте разные контексты исполнения для одного типа вызова:

function nonStrictSum(a, b) {

// non-strict mode

console.log(this === window); // => true

return a + b;

}

function strictSum(a, b) {

'use strict';

// strict mode is enabled

console.log(this === undefined); // => true

return a + b;

}

// nonStrictSum() is invoked as a function in non-strict mode

// this in nonStrictSum() is the window object

console.log(nonStrictSum(5, 6)); // => 11

// strictSum() is invoked as a function in strict mode

// this in strictSum() is undefined

console.log(strictSum(8, 12)); // => 20

Ловушка: this во внутренней функции

Обычной ошибкой при работе с вызовом функции является уверенность в том, что this во внутренней функции такой же, как и во внешней.

Вообще-то контекст внутренней функции зависит только от вызова, а не от контекста внешней функции.

Чтобы получить ожидаемый this, модифицируйте контекст внутренней функции при помощи непрямого вызова (используя [.call()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call) или [.apply()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply), об этом позже) или создайте связанную функцию (используя [.bind()](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_objects/Function/bind), об этом тоже поговорим позже).

Следующий пример вычисляет сумму двух чисел:

var numbers = {

numberA: 5,

numberB: 10,

sum: function() {

console.log(this === numbers); // => true

function calculate() {

// this is window or undefined in strict mode

console.log(this === numbers); // => false

return this.numberA + this.numberB;

}

return calculate();

}

};

console.log(numbers.sum()); // => NaN or throws TypeError in strict mode

numbers.sum() — это вызов метода объекта, поэтому контекстом sum является объект numbers. Функция calculate определена внутри sum, поэтому вы можете ожидать, что this — это объект numbers и в calculate(). Тем не менее, calculate() — это вызов функции, а не метода, и поэтому его this — это глобальный объект window или undefined в strict mode. Даже если контекстом внешней функции sum является объект numbers, у него здесь нет власти.

Результатом вызова numbers.sum() является NaN или ошибка TypeError: Cannot read property 'numberA' of undefined в strict mode. Точно не ожидаемый результат 5 + 10 = 15, а всё потому, что calculate вызвана некорректно.

Для решения проблемы функция calculate должна быть исполнена в том же контексте, что и метод sum, чтобы получить доступ к значениям numberA и numberB. Это можно сделать при помощи метода .call():

var numbers = {

numberA: 5,

numberB: 10,

sum: function() {

console.log(this === numbers); // => true

function calculate() {

console.log(this === numbers); // => true

return this.numberA + this.numberB;

}

// use .call() method to modify the context

return calculate.call(this);

}

};

console.log(numbers.sum()); // => 15

calculate.call(this) исполняет функцию calculate, но дополнительно модифицирует контекст в соответствии с первым параметром. Теперь this.numberA + this.numberB эквивалентно numbers.numberA + numbers.numberB и функция возвращает ожидаемый результат 5 + 10 = 15.

Вызов метода

Метод — это функция, хранящаяся в объекте. Пример:

var myObject = {

// helloFunction is a method

helloFunction: function() {

return 'Hello World!';

}

};

var message = myObject.helloFunction();

console.log(message);

helloFunction — это метод в myObject. Для доступа к методу нужно использовать аксессор: myObject.helloFunction.

**Вызов метода** совершается, когда за выражением в виде аксессора, расценивающемся как объект функции, следует пара скобок и разделенный запятыми список аргументов между ними.

В прошлом примере myObject.helloFunction() — это вызов метода helloFunction объекта myObject. Также вызовами метода являются: [1, 2].join(',') или /\s/.test('beautiful world').

Важно отличать вызов функции от вызова метода. Главным отличием является то, что для вызова метода необходим аксессор (<expression>.functionProperty() или <expression>['functionProperty']()), а для вызова функции — нет (<expression>()).

console.log(

['Hello', 'World'].join(', ') // method invocation

);

console.log(

({ ten: function() { return 10; } }).ten() // method invocation

);

var obj = {};

obj.myFunction = function() {

return new Date().toString();

};

console.log(

obj.myFunction() // method invocation

);

var otherFunction = obj.myFunction;

console.log(otherFunction()); // function invocation

console.log(parseFloat('16.60')); // function invocation

console.log(isNaN(0)); // function invocation

this при вызове метода

this — это объект, которому принадлежит метод

При вызове метода, принадлежащего объекту, this становится этим объектом.

Давайте создадим объект, метод которого увеличивает число на 1:

var calc = {

num: 0,

increment: function() {

console.log(this === calc); // => true

this.num += 1;

return this.num;

}

};

// method invocation. this is calc

console.log(calc.increment()); // => 1

console.log(calc.increment()); // => 2

Вызов calc.increment() сделает контекстом функции increment объект calc. Поэтому можно спокойно использовать this.num.

Объект JavaScript наследует метод своего прототипа. Когда вызывается метод, унаследованный от объекта, контекстом всё равно является сам объект:

var myDog = Object.create({

sayName: function() {

console.log(this === myDog); // => true

return this.name;

}

});

myDog.name = 'Milo';

// method invocation. this is myDog

console.log(myDog.sayName()); // => 'Milo'

[Object.create()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create) создаёт новый объект myDog и создаёт прототип. Объект myDog наследует метод sayName. Когда исполняется myDog.sayName(), myDog является контекстом исполнения.

В синтаксисе ECMAScript 6 [class](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes" \t "_blank) контекст вызова метода — тоже сам объект:

/\* jshint esnext: true \*/

class Planet {

constructor(name) {

this.name = name;

}

getName() {

console.log(this === earth); // => true

return this.name;

}

}

var earth = new Planet('Earth');

// method invocation. the context is earth

console.log(earth.getName()); // => 'Earth'

Ловушка: отделение метода от его объекта

Метод объекта можно переместить в отдельную переменную. При вызове метода с использованием этой переменной вы можете подумать, что this — это объект, в котором определён метод.

На самом деле, если метод вызван без объекта, происходит вызов функции, и this становится глобальным объектом window или undefined. Создание связанной функции исправляет контекст — им становится объект, в котором содержится метод.

Следующий пример создаёт конструктор Animal и его экземпляр — myCat. Затем через 1 секунду setTimeout() логирует информацию об объекте myCat:

function Animal(type, legs) {

this.type = type;

this.legs = legs;

this.logInfo = function() {

console.log(this === myCat); // => false

console.log('The ' + this.type + ' has ' + this.legs + ' legs');

};

}

var myCat = new Animal('Cat', 4);

// logs "The undefined has undefined legs"

// or throws a TypeError, in strict mode

setTimeout(myCat.logInfo, 1000);

Вы можете подумать, что setTimeout вызовет myCat.logInfo(), которая запишет информацию об объекте myCat. Но метод отделяется от объекта, когда передаётся в качестве параметра: setTimout(myCat.logInfo), и через секунду происходит вызов функции. Когда logInfo вызывается как функция, this становится глобальным объектом или undefined (но не объектом myCat), поэтому информация об объекте выводится некорректно.

Функцию можно связать с объектом, используя метод [.bind()](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_objects/Function/bind). Если отделённый метод связан с объектом myCat, проблема контекста решается:

function Animal(type, legs) {

this.type = type;

this.legs = legs;

this.logInfo = function() {

console.log(this === myCat); // => true

console.log('The ' + this.type + ' has ' + this.legs + ' legs');

};

}

var myCat = new Animal('Cat', 4);

// logs "The Cat has 4 legs"

setTimeout(myCat.logInfo.bind(myCat), 1000);

myCat.logInfo.bind(myCat) возвращает новую функцию, исполняемую в точности как logInfo, но this которой остаётся myCat даже в случае вызова функции.

Вызов конструктора

**Вызов конструктора** совершается, когда за ключевым словом [new](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/new" \t "_blank) следует выражение, расцениваемое как объект функции, и пара скобок с разделённым запятыми списком аргументов. Пример: new RegExp('\\d').

В этом примере объявляется функция Country, которая затем вызывается в качестве конструктора:

function Country(name, traveled) {

this.name = name ? name : 'United Kingdom';

this.traveled = Boolean(traveled); // transform to a boolean

}

Country.prototype.travel = function() {

this.traveled = true;

};

// Constructor invocation

var france = new Country('France', false);

// Constructor invocation

var unitedKingdom = new Country;

france.travel(); // Travel to France

new Country('France', false) — это вызов конструктора функции Country. Результатом исполнения является новые объект, чьё поле name равняется 'France'.

Если конструктор вызван без аргументов, скобки можно опустить: new Country.

Начиная с ECMAScript 6, JavaScript позволяет определять конструкторы ключевым словом [class](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes" \t "_blank):

/\* jshint esnext: true \*/

class City {

constructor(name, traveled) {

this.name = name;

this.traveled = false;

}

travel() {

this.traveled = true;

}

}

// Constructor invocation

var paris = new City('Paris', false);

paris.travel();

new City('Paris') — это вызов конструктора. Инициализация объекта управляется специальным методом класса: [constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor" \t "_blank), this которого является только что созданным объектом.

Вызов конструктора создаёт новый пустой объект, наследующий свойства от прототипа конструктора. Ролью функции-конструктора является инициализация объекта. Как вы уже знаете, контекст этого типа вызова называется экземпляром. Это — тема следующей главы.

Когда перед аксессором myObject.myFunction идёт ключевое слово new, JavaScript совершит вызов конструктора, а не метода. Возьмём в качестве примера new myObject.myFunction(): сперва при помощи аксессора extractedFunction = myObject.myFunction функция извлекается, а затем вызывается как конструктор для создания нового объекта: new extractedFunction().

this в вызове конструктора

this — это только что созданный объект

Контекстом вызова конструктора является только что созданный объект. Он используется для инициализации объекта данными из аргументом функции-конструктора.

Давайте проверим контекст в следующем примере:

function Foo () {

console.log(this instanceof Foo); // => true

this.property = 'Default Value';

}

// Constructor invocation

var fooInstance = new Foo();

console.log(fooInstance.property); // => 'Default Value'

new Foo() делает вызов конструктора с контекстом fooInstance. Объект инициализируется внутри Foo: this.property задаётся значением по умолчанию.

Тоже самое происходит при использовании [class](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes" \t "_blank), только инициализация происходит в методе constructor:

/\* jshint esnext: true \*/

class Bar {

constructor() {

console.log(this instanceof Bar); // => true

this.property = 'Default Value';

}

}

// Constructor invocation

var barInstance = new Bar();

console.log(barInstance.property); // => 'Default Value'

Когда исполняется new Bar(), JavaScript создаёт пустой объект и делает его контекстом метода constructor. Теперь вы можете добавлять свойства, используя this: this.property = 'Default Value'.

Ловушка: как не забыть про new

Некоторые функции JavaScript создают экземпляры при вызове не только в качестве конструктора, но и функции. Например, RegExp:

var reg1 = new RegExp('\\w+');

var reg2 = RegExp('\\w+');

console.log(reg1 instanceof RegExp); // => true

console.log(reg2 instanceof RegExp); // => true

console.log(reg1.source === reg2.source); // => true

При исполнении new RegExp('\\w+') и RegExp('\\w+') JavaScript создаёт эквивалентные объекты регулярных выражений.

Использование вызова функции для создания объектом потенциально опасно (если опустить [фабричный метод](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B1%D1%80%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))), потому что некоторые конструкторы могут не инициализировать объект при отсутствии ключевого слова new.

Следующий пример иллюстрирует проблему:

function Vehicle(type, wheelsCount) {

this.type = type;

this.wheelsCount = wheelsCount;

return this;

}

// Function invocation

var car = Vehicle('Car', 4);

console.log(car.type); // => 'Car'

console.log(car.wheelsCount); // => 4

console.log(car === window); // => true

Vehicle — это функция, задающая свойства type и wheelsCount объекту-контексту. При исполнении Vehicle('Car', 4) возвращается объект car, обладающий корректными свойствами: car.type равен 'Car' а car.wheelsCount — 4. Легко подумать, что всё работает как надо.

Тем не менее, this — это объект window при вызове функции, и Vehicle('Car', 4) задаёт свойства объекта window — упс, что-то пошло не так. Новый объект не создан.

Обязательно используйте оператор new, когда ожидается вызов конструктора:

function Vehicle(type, wheelsCount) {

if (!(this instanceof Vehicle)) {

throw Error('Error: Incorrect invocation');

}

this.type = type;

this.wheelsCount = wheelsCount;

return this;

}

// Constructor invocation

var car = new Vehicle('Car', 4);

console.log(car.type); // => 'Car'

console.log(car.wheelsCount); // => 4

console.log(car instanceof Vehicle); // => true

// Function invocation. Generates an error.

var brokenCar = Vehicle('Broken Car', 3);

new Vehicle('Car', 4) работает верно: новый объект создан и инициализирован, поскольку присутствует слово new.

В вызове функции добавлена верификация: this instanceof Vehicle, чтобы убедиться, что у контекста исполнения верный тип объекта. Если this — не Vehicle, генерируется ошибка. Таким образом, если исполняется  Vehicle('Broken Car', 3) (без new), то выбрасывается исключение: Error: Incorrect invocation.

Непрямой вызов

**Непрямой вызов** производится, когда функция вызывается методами .call() или .apply().

Функции в JavaScript — [объекты первого класса](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_%D0%BF%D0%B5%D1%80%D0%B2%D0%BE%D0%B3%D0%BE_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%B0), то есть функция — это объект типа [Function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function" \t "_blank).

Из [списка методов](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function#Methods) этой функции два, [.call()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call) и [.apply()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply), используются для вызова функции с настраиваемым контекстом:

* Метод .call(thisArg[, arg1[, arg2[, ...]]]) принимает в качестве первого аргумента thisArg контекст вызова, а список аргументов arg1, arg2, ... передаётся вызываемой функции.
* Метод .apply(thisArg, [args]) принимает в качестве первого аргумента thisArg контекст вызова, а [array-like объект](http://www.2ality.com/2013/05/quirk-array-like-objects.html" \t "_blank) [args] передаётся вызываемой функции в качестве аргумента.

Следующий пример демонстрирует непрямой вызов:

function increment(number) {

return ++number;

}

console.log(increment.call(undefined, 10)); // => 11

console.log(increment.apply(undefined, [10])); // => 11

increment.call() и increment.apply() оба вызывают функцию-инкремент с аргументом 10.

Главным отличием между ними является то, что .call() принимает список аргументов, например, myFunction.call(thisValue, 'value1', 'value2'), а .apply() принимает эти значения в виде array-like объекта: myFunction.apply(thisValue, ['value1', 'value2']).

this при непрямом вызове

this — это первый аргумент .call() или .apply()

Очевидно, что при непрямом вызове this — значение, передаваемое .call() или .apply() в качестве первого аргумента. Пример:

var rabbit = { name: 'White Rabbit' };

function concatName(string) {

console.log(this === rabbit); // => true

return string + this.name;

}

// Indirect invocations

console.log(concatName.call(rabbit, 'Hello ')); // => 'Hello White Rabbit'

console.log(concatName.apply(rabbit, ['Bye '])); // => 'Bye White Rabbit'

Непрямой вызов может пригодиться, когда функцию нужно вызвать в особом контексте, например, решить проблему при вызове функции, где this — всегда window или undefined. Его также можно использовать для симуляции вызова метода объекта.

Ещё одним примером использования является создание иерархии классов в ES5 для вызова родительского конструктора:

function Runner(name) {

console.log(this instanceof Rabbit); // => true

this.name = name;

}

function Rabbit(name, countLegs) {

console.log(this instanceof Rabbit); // => true

// Indirect invocation. Call parent constructor.

Runner.call(this, name);

this.countLegs = countLegs;

}

var myRabbit = new Rabbit('White Rabbit', 4);

console.log(myRabbit); // { name: 'White Rabbit', countLegs: 4 }

Runner.call(this, name) в Rabbit создаёт непрямой вызов родительской функции для инициализации объекта.

Связанная функция

**Связанная функция** — это функция, связанная с объектом. Обычно она создаётся из обычной функции при помощи метода [.bind()](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_objects/Function/bind). У двух функций совпадают тела и области видимости, но различаются контексты.

Метод .bind(thisArg[, arg1[, arg2[, ...]]]) принимает в качестве первого аргумента thisArg контекст вызова связанной функции, а необязательный список аргументов arg1, arg2, ... передаётся вызываемой функции. Он возвращает новую функцию, связанную с thisArg.

Следующий код создаёт связанную функцию и вызывает её:

function multiply(number) {

'use strict';

return this \* number;

}

// create a bound function with context

var double = multiply.bind(2);

// invoke the bound function

console.log(double(3)); // => 6

console.log(double(10)); // => 20

multiply.bind(2) возвращает новый объект функции double, который связан с числом 2. Код и область видимости у multiply и double совпадают.

В отличие от методов .apply() и .call(), сразу вызывающих функцию, метод .bind() возвращает новую функцию, которую впоследствии нужно будет вызвать с уже заданным this.

this в связанной функции

this — это первый аргумент .bind()

Ролью .bind() является создание новой функции, чей вызов будет иметь контекст, заданный в первом аргументе .bind(). Это — мощный инструмент, позволяющий создавать функции с заранее определённым значением this.

Давайте посмотрим, как настроить this связанной функции:

var numbers = {

array: [3, 5, 10],

getNumbers: function() {

return this.array;

}

};

// Create a bound function

var boundGetNumbers = numbers.getNumbers.bind(numbers);

console.log(boundGetNumbers()); // => [3, 5, 10]

// Extract method from object

var simpleGetNumbers = numbers.getNumbers;

console.log(simpleGetNumbers()); // => undefined or throws an error in strict mode

numbers.getNumbers.bind(numbers) возвращает функцию boundGetNumbers, которая связана с объектом numbers. Затем boundGetNumbers() вызывается с this, равным numbers, и возвращает корректный объект.

Функцию numbers.getNumbers можно извлечь в переменную simpleGetNumbers и без связывания. При дальнейшем вызове функции simpleGetNumbers() задаёт this как window или undefined, а не numbers. В этом случае simpleGetNumbers() не вернет корректное значение.

.bind() создаёт перманентную контекстную ссылку и хранит её. Связанная функция не может изменить контекст, используя .call() или .apply() с другим контекстом — даже повторное связывание не даст эффекта.

Только вызов связанной функции как конструктора может изменить контекст, но это не рекомендуется (используйте нормальные функции).

В следующем примере сперва объявляется связанная функция, а затем производится попытка изменить контекст:

function getThis() {

'use strict';

return this;

}

var one = getThis.bind(1);

// Bound function invocation

console.log(one()); // => 1

// Use bound function with .apply() and .call()

console.log(one.call(2)); // => 1

console.log(one.apply(2)); // => 1

// Bind again

console.log(one.bind(2)()); // => 1

// Call the bound function as a constructor

console.log(new one()); // => Object

Только new one() изменяет контекст связанной функции, в остальных типах вызова this всегда равен 1.

Стрелочная функция

**Стрелочная функция** нужна для более короткой формы объявления функции и [лексического](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BB%D0%B0%D1%81%D1%82%D1%8C_%D0%B2%D0%B8%D0%B4%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B8#.D0.9B.D0.B5.D0.BA.D1.81.D0.B8.D1.87.D0.B5.D1.81.D0.BA.D0.B8.D0.B5_vs._.D0.B4.D0.B8.D0.BD.D0.B0.D0.BC.D0.B8.D1.87.D0.B5.D1.81.D0.BA.D0.B8.D0.B5_.D0.BE.D0.B1.D0.BB.D0.B0.D1.81.D1.82.D0.B8_.D0.B2.D0.B8.D0.B4.D0.B8.D0.BC.D0.BE.D1.81.D1.82.D0.B8) связывания контекста.

Её можно использовать следующим образом:

/\* jshint esnext: true \*/

var hello = (name) => {

return 'Hello ' + name;

};

console.log(hello('World')); // => 'Hello World'

// Keep only even numbers

console.log([1, 2, 5, 6].filter(item => item % 2 === 0)); // => [2, 6]

Стрелочные функции используют облегчённый синтаксис, убирая ключевое слово function. Можно даже опустить return, когда у функции есть лишь одно выражение.

Стрелочная функция анонимна [анонимна](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/name" \t "_blank), что означает, что её свойство name — пустая строка ''. Таким образом, у неё нет лексического имени, которое нужно для рекурсии и управления хэндлерами.

Кроме того, она не предоставляет объект [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments" \t "_blank), в отличие от обычной функции. Тем не менее, это можно исправить, используя [rest-параметры](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/rest_parameters" \t "_blank) ES6:

/\* jshint esnext: true \*/

var sumArguments = (...args) => {

console.log(typeof arguments); // => 'undefined'

return args.reduce((result, item) => result + item);

};

console.log(sumArguments.name); // => ''

console.log(sumArguments(5, 5, 6)); // => 16

this в стрелочной функции

this — это контекст, в котором определена стрелочная функция

Стрелочная функция не создаёт свой контекст исполнения, а заимствует this из внешней функции, в которой она определена.

Следующий пример показывает прозрачность контекста:

/\* jshint esnext: true \*/

class Point {

constructor(x, y) {

this.x = x;

this.y = y;

}

log() {

console.log(this === myPoint);

setTimeout(()=> {

console.log(this === myPoint); // => true

console.log(this.x + ':' + this.y); // => '95:165'

}, 1000);

}

}

var myPoint = new Point(95, 165);

myPoint.log();

setTimeout вызывает стрелочную функцию в том же контексте (метод myPoint), что и метод log(). Как мы видим, стрелочная функция «наследует» контекст той функции, в которой определена.

Если попробовать использовать в этом примере обычную функцию, она создаст свой контекст (window или undefined). Поэтому для того, чтобы код работал корректно, нужно вручную привязать контекст: setTimeout(function() {...}.bind(this)). Это громоздко, поэтому проще использовать стрелочную функцию.

Если стрелочная функция определена вне всех функций, её контекст — глобальный объект:

/\* jshint esnext: true \*/

var getContext = () => {

console.log(this === window); // => true

return this;

};

console.log(getContext() === window); // => true

Стрелочная функция связывается с лексическим контекстом **раз и навсегда**. this нельзя изменить даже при помощи метод смены контекста:

/\* jshint esnext: true \*/

var numbers = [1, 2];

(function() {

var get = () => {

return this;

};

console.log(this === numbers); // => true

console.log(get()); // => [1, 2]

// Use arrow function with .apply() and .call()

console.log(get.call([0])); // => [1, 2]

console.log(get.apply([0])); // => [1, 2]

// Bind

console.log(get.bind([0])()); // => [1, 2]

}).call(numbers);

Функция, вызываемая непрямым образом с использованием .call(numbers), задаёт this значение numbers.  Стрелочная функция get также получает numbers в качестве this, поскольку принимает контекст лексически. Неважно, как вызывается get, её контекстом всегда будет numbers. Непрямой вызов с другим контекстом (используя .call() или .apply()), повторное связывание (с использованием .bind()) не принесут эффекта.

Стрелочную функцию нельзя использовать в качестве конструктора. Если вызвать new get(), JavaScript выбросит ошибку: TypeError: get is not a constructor.

Ловушка: определение метода стрелочной функцией

Вы можете захотеть использовать стрелочную функцию для объявления метода. Справедливо: их объявления гораздо короче по сравнению с обычным выражением: (param) => {...} вместо function(param) {..}.

В этом примере демонстрируется определение метода format() класса Period с использованием стрелочной функции:

/\* jshint esnext: true \*/

function Period (hours, minutes) {

this.hours = hours;

this.minutes = minutes;

}

Period.prototype.format = () => {

console.log(this === window); // => true

return this.hours + ' hours and ' + this.minutes + ' minutes';

};

var walkPeriod = new Period(2, 30);

console.log(walkPeriod.format());

Так как format — стрелочная функция, определённая в глобальном контексте, её this — это объект window. Даже если format исполняется в качестве метода объекта walkPeriod.format(), window остаётся контекстом вызова. Так происходит, потому что стрелочная функция имеет статический контекст, не изменяемый другими типами вызовов.

this — это window, поэтому this.hours и this.minutes становятся undefined. Метод возвращает строку 'undefined hours and undefined minutes', что не является желаемым результатом.

Функциональное выражение решает проблему, поскольку обычная функция изменяет свой контекст в зависимости от вызова:

function Period (hours, minutes) {

this.hours = hours;

this.minutes = minutes;

}

Period.prototype.format = function() {

console.log(this === walkPeriod); // => true

return this.hours + ' hours and ' + this.minutes + ' minutes';

};

var walkPeriod = new Period(2, 30);

console.log(walkPeriod.format());

walkPeriod.format() — это вызов метода с контекстом walkPeriod. this.hours принимает значение 2, а this.minutes — 30, поэтому метод возвращает корректный результат: '2 hours and 30 minutes'.

Заключение

Поскольку вызов функции имеет наибольшее влияние на this, отныне **не спрашивайте**:

Откуда берется this?

а **спрашивайте**:

Как функция вызывается?

А в случае со стрелочной функцией спросите:

Каков this там, где объявлена стрелочная функция?

Такой подход к this убережет вас от лишней головной боли.

Не путайтесь в контекстах! 🙂