**Всё о this. Введение**

В одном из прошлых спринтов вы впервые столкнулись с ключевым словом this. В этой теме мы расскажем всё о this, но сначала напомним, что вы уже изучили.

Если функцию вызывают как метод объекта, this содержит ссылку на объект, на котором она вызвана:

Скопировать кодJAVASCRIPT

const obj = {

prop: 'Свойство',

method: function () {

console.log(this.prop);

}

}

obj.method(); *// "Свойство"*

Вызов функции как метода объекта — не единственный способ установки this*,* есть и другие*.* Если не разобраться в них, механизм работы this может показаться непредсказуемым. Может быть, поэтому о this так часто спрашивают на собеседованиях.

В этой теме мы постараемся разобраться с этим всем раз и навсегда.

# Строгий режим

JavaScript создавали как язык для новичков. Поэтому в нём есть механизмы для исправления ошибок, которые делает разработчик:

Скопировать кодJAVASCRIPT

let greeting = 'Привет';

const age = Number(prompt('Введите возраст: ', 0));

if (age > 18) {

*// Опечатались в имени переменной:*

*// написали greting вместо greeting*

greting = 'Здравствуйте';

}

Что такое prompt?

Мы опечатались, но если запустить этот код, консоль не покажет ошибку. Движок «догадался», что мы забыли объявить переменную greting, и объявил её самостоятельно в глобальной области видимости. В глазах JavaScript код стал выглядеть так:

Скопировать кодJAVASCRIPT

let greeting = 'Привет';

const age = Number(prompt('Введите возраст: ', 0));

if (age > 18) {

let greting; *// Движок самостоятельно объявил переменную*

greting = 'Здравствуйте';

}

Движок нас неправильно понял: он «подумал», что мы забыли объявить переменную, прежде чем присваивать ей значение. На самом деле мы опечатались. Так что движок оказал нам медвежью услугу: теперь найти ошибку труднее.

В спецификации ES5 появилась возможность отказаться от такой помощи со стороны движка. Для этого нужно писать код в строгом режиме. Чтобы его включить, напишите в начале файла 'use strict':

Скопировать кодJAVASCRIPT

'use strict';

let greeting = 'Привет';

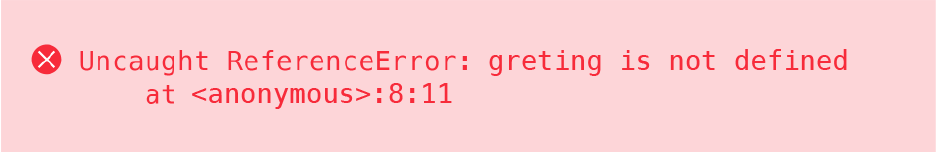
const age = Number(prompt('Введите возраст: ', 0));

if (age > 18) {

greting = 'Здравствуйте';

}

Теперь браузер сообщает об ошибке, так что её легко найти и исправить:



Строгий режим как бы говорит браузеру выдавать ошибки, на которые тот раньше закрывал глаза.

# Какие изменения несёт строгий режим

В этом уроке расскажем о части отличий строгого режима от обычного. Об остальных поговорим в других уроках темы.

1. Значение контекста this. Вы уже [знаете](https://praktikum.yandex.ru/trainer/web/lesson/1a1651c3-b5b3-4db0-8f42-1ce705395f25/task/abd6a3ae-6cf2-40c9-9ef0-1554e11dad66/) про привязку this по умолчанию. В зависимости от наличия строго режима this принимает одно из двух значений:
   * undefined — в строгом режиме,
   * window — в обычном.

Скопировать кодJAVASCRIPT

function globalFunction() {

'use strict';

console.log(this);

}

globalFunction(); *// undefined — в строгом режиме this по умолчанию undefined*

Строгий режим задаёт this равным undefined для функций, вызванных по имени. Привязку по умолчанию редко используют намеренно — обычно это следствие какой-то ошибки или так называемой потери контекста. О том, что это такое, расскажем в следующем уроке.

1. Если переменная не объявлена, в неё нельзя записать значение:

Скопировать кодJAVASCRIPT

'use strict';

*/\* переменную нужно объявить перед записью*

*значения в неё, иначе будет ошибка \*/*

greeting = 'Привет'; *// ошибка*

1. В теме о функциях мы рассказывали о переменной arguments: она создаётся внутри функции при объявлении и содержит все переданные аргументы. В обычном режиме мы могли менять значения аргументов, обращаясь к ним по индексам:

Скопировать кодJAVASCRIPT

function consoleDog(dog) {

arguments[0] = 'Чихуахуа'; *// Перезаписываем аргумент*

console.log(dog);

}

consoleDog('Джек Рассел'); *// "Чихуахуа" — результат изменился*

Строгий режим такого не допустит:

Скопировать кодJAVASCRIPT

'use strict';

function consoleDog(dog) {

arguments[0] = 'Чихуахуа'; *// попытаемся изменить первый аргумент*

console.log(dog);

}

consoleDog('Джек Рассел'); *// "Джек Рассел" — аргумент не перезаписался*

1. В строгом режиме параметры функции не могут иметь одинаковые имена — консоль покажет синтаксическую ошибку:

Скопировать кодJAVASCRIPT

'use strict';

function consoleDog(dog, dog) { *// SyntaxError*

console.log(dog);

}

# Как использовать

Если написать 'use strict'; в начале файла, весь его код будет исполняться в строгом режиме:

Скопировать кодJAVASCRIPT

*/\* script.js \*/*

'use strict';

*// здесь будет строгий режим*

function callMe() {

*// и здесь будет строгий режим*

}

Строгий режим можно включить для отдельной функции. Для этого следует написать 'use strict'; в начале тела этой функции:

Скопировать кодJAVASCRIPT

function callMe() {

'use strict';

*// здесь будет строгий режим*

}

*// а здесь нет*

Строгий режим нельзя отменить. В языке нет специальной директивы, которая бы выключала его. Единственный способ отключить строгий режим — убрать директиву 'use strict';.

# Зачем нужен новый режим?

«Медвежьи услуги» движка — это ошибки проектирования. Когда создавали JavaScript, хотели как лучше, а получилось совсем наоборот. Было бы здорово вернуться в прошлое и поменять стандарт. Но сейчас такого сделать уже нельзя.

До того как был придуман строгий режим, разработчики опирались на логику работы обычного. Если сейчас выбросить обычный режим из спецификации, сайты, сделанные по старому стандарту, сломаются. Поэтому нам следует заявлять прямо о желании следовать новому стандарту:

Скопировать кодJAVASCRIPT

'use strict';

Значит ли это, что теперь в начале каждого файла нужно писать 'use strict';? Нет, так как в прошлом спринте вы начали разбивать код на модули. В модулях строгий режим включён по умолчанию.

# Вызов метода объекта

В прошлом уроке мы рассматривали такую функцию:

Скопировать кодJAVASCRIPT

function globalFunction() {

'use strict';

console.log(this);

}

globalFunction(); *// undefined*

Сейчас код падает. Но мы можем исправить эту ошибку. Обратимся ко второму правилу, которому подчиняется this: это вызов метода объекта или неявная привязка. У этого правила приоритет выше, чем у привязки по умолчанию, но ниже, чем у двух других правил.

Скопировать кодJSX

window.myData = 'Important data'

function globalFunction() {

'use strict';

console.log(this.myData);

}

window.globalFunction(); *// 'Important data'*

Обратите внимание, что мы изменили точку вызова функции. Раньше мы вызывали её из глобальной области видимости, а теперь — как метод объекта window. Так мы неявно указали контекст — window.

Правило привязки по умолчанию из прошлого урока имеет самый низкий приоритет среди всех остальных. Поэтому срабатывает правило неявной привязки, а привязка по умолчанию игнорируется. Всегда изучайте точку вызова функции, чтобы понять, какие правила привязки this применяются.

Ещё пример:

Скопировать кодJAVASCRIPT

const counter = {

count: 0,

increment() {

this.count++;

console.log('Count: ' + this.count);

}

};

counter.increment(); *// Count: 1*

При использовании неявной привязки определить контекст this очень просто: посмотрите, что находится слева от точки перед именем функции, — это и есть this. У нас в примере this ссылается на объект counter, потому что функция increment вызвана как метод объекта — через точку. Так происходит неявная привязка к этому объекту.

## Потеря контекста

Вы уже [знаете](https://praktikum.yandex.ru/trainer/web/lesson/3fa8a986-6ba3-44e0-a741-be77bc66619c/task/b73ced6d-e0ec-4c02-b524-675db3fc9f83/) про такое явление, как потеря контекста и как строгий режим влияет на контекст. Посмотрите на этот код:

Скопировать кодJAVASCRIPT

const user = {

username: 'Peter',

auth() {

console.log(`${this.username} has logged in`);

}

}

const adminAuth = user.auth;

adminAuth(); *// undefined has logged in*

На первый взгляд кажется, что должно было сработать правило неявной привязки и adminAuth ссылается на user.auth. Но это не так. Посмотрите на точку вызова функции — это глобальная область видимости. В момент вызова движок определяет this по первому правилу — привязке по умолчанию. Получается, в this оказывается глобальный объект window, у которого нет свойства username. Поэтому нам возвращается undefined.

В строгом режиме этот код приводит к ошибке, потому что значение this равно undefined. Обращение к свойству undefined.username ломает код:

Скопировать кодJAVASCRIPT

'use strict';

const user = {

username: 'Peter',

auth() {

console.log(`${this.username} has logged in`);

}

};

const adminAuth = user.auth;

adminAuth(); *// TypeError: Cannot read property 'username' of undefined*

Чтобы не терять контекст, его можно указать явно методами call, apply или bind. Про bind вы уже [знаете](https://praktikum.yandex.ru/trainer/web/lesson/1a1651c3-b5b3-4db0-8f42-1ce705395f25/task/abd6a3ae-6cf2-40c9-9ef0-1554e11dad66/). А call и apply рассмотрим в следующем уроке.

# Явная привязка. Методы call, apply и bind

В отличие от известного вам метода bind методы call и apply вызывают функцию. Рассмотрим каждый из этих методов.

# Метод call

Метод call вызывает функцию, явно указывая её контекст. То есть при вызове функции мы явно сообщаем движку, какой объект должен быть записан в this:

Скопировать кодJAVASCRIPT

const user = {

username: 'Peter',

auth() {

console.log(`${this.username} has logged in`);

}

};

const adminAuth = user.auth;

adminAuth.call(user); *// Peter has logged in*

Метод call вызывает функцию, он принимает сколько угодно аргументов:

* первый аргумент — это контекст, то есть объект, который будет записан в this. В примере это объект user;
* все аргументы, начиная со второго, это параметры самой функции, которую мы вызываем:

Скопировать кодJAVASCRIPT

const user = {

username: 'Peter',

auth(greeting) { *// у функции теперь есть параметр greeting*

console.log(`${greeting} ${this.username}`);

}

};

const adminAuth = user.auth;

adminAuth.call(user, 'Hello'); *// Hello Peter*

Этих аргументов может быть сколько угодно:

Скопировать кодJAVASCRIPT

const user = {

username: 'Peter',

auth() {

console.log(arguments); *// выведем аргументы в консоль*

}

};

const adminAuth = user.auth;

adminAuth.call(user, 1, 2, 3, 4, 5); *// Arguments(5) [1, 2, 3, 4, 5]*

# Метод apply

Метод apply делает всё то же самое, что и call, то есть явно указывает значение this, но по-другому принимает аргументы. Методу apply первым аргументом передают контекст, а параметры функции — вторым аргументом в массиве:

Скопировать кодJAVASCRIPT

const car = {

registrationNumber: 'O287AE',

brand: 'Tesla'

};

function displayDetails(greeting, ownerName) {

console.log(`${greeting} ${ownerName}`);

console.log(`Car info: ${this.registrationNumber} ${this.brand}`);

}

displayDetails.apply(car, ['Hello', 'Matt']);

*/\**

*Hello Matt*

*Car info: O287AE Tesla*

*\*/*

# Привязка new

Мы уже разобрали 3 привязки из 4: привязку по умолчанию, неявную привязку и явную. Остался четвёртый способ задать значение this: при вызове функции с оператором new. Напомним, что происходит при вызове функции с new:



Оператор new используют, когда функцию нужно вызвать как конструктор:

Скопировать кодJAVASCRIPT

function Human(name) {

this.name = name;

}

const bob = new Human('Bob');

console.log(bob) *// Human { name: 'Bob' }*

Когда функцию вызывают с оператором new, движок автоматически выполняет три действия:

1. Конструирует новый объект.
2. Устанавливает этот объект в значение this для вызванной функции.
3. Возвращает this.

Если вызвать функцию с new, this возвращается автоматически. Функция может ничего не возвращать или возвращать примитивное значение, но при её вызове с new всё равно вернётся this.

Но есть одно исключение. Если функция возвращает объект, вызов функции с new вернёт этот объект:

Скопировать кодJAVASCRIPT

function Plane(model) {

this.model = model;

return { model: 'TU-134' };

}

const airbus = new Plane('Airbus');

console.log(airbus); *// { model: 'TU-134' }*

# Оператор new и методы call, apply, bind

Оператор new не работает вместе с методами call, apply и bind:

Скопировать кодJAVASCRIPT

function Driver(name, category) {

this.name = name;

this.category = category;

}

const Henry = new Driver.call(window, 'Henry', 'A'); *// TypeError: Driver.call is not a constructor*

const Peter = new Driver.apply(window, ['Peter', 'B']); *// TypeError: Driver.apply is not a constructor*

const Matt = new Driver.bind(window, 'Matt', 'C')(); *// TypeError: Driver.bind is not a constructor*

Но мы можем явно указывать контекст для функции в уже созданном объекте:

Скопировать кодJAVASCRIPT

function Windows(name) {

this.name = 'Windows 10';

this.printInfo = function () {

console.log('Your OS: ' + this.name);

};

}

const Linux = {

name: 'Ubuntu',

};

const theOs = new Windows();

theOs.printInfo(); *// Your OS: Windows 10*

theOs.printInfo.call(Linux); *// Your OS: Ubuntu*

theOs.printInfo.apply(Linux); *// Your OS: Ubuntu*

theOs.printInfo.bind(Linux)(); *// Your OS: Ubuntu*

# Стрелочные функции

Вы уже знакомы со стрелочными функциями. Кроме более короткого синтаксиса у них есть ещё одна важная особенность.

Их механизм привязки this отличается от четырёх рассмотренных способов. Ключевое слово this стрелочных функций привязывается не во время вызова, а во время объявления.

# Отсутствие собственного значения this

Стрелочные функции берут значение this из замыкания. В них this такой же, как и this внешней области видимости. Из-за этого нет смысла использовать стрелочную функцию как метод объекта:

Скопировать кодJAVASCRIPT

const myObject = {

myMethod: () => {

console.log(this);

}

};

myObject.myMethod(); *// window (или undefined в строгом режиме)*

# Использование с call, apply или bind

Поскольку this у стрелочной функции определяется только местом её объявления, вызов стрелочных функций методами call или apply никак не влияет на значение this. Метод bind тоже не сработает:

Скопировать кодJAVASCRIPT

function Frontender() {

this.profession = 'Frontend Developer';

this.printProfession = () => {

console.log('I am a ' + this.profession);

};

}

const backender = {

profession: 'Backend Developer'

};

const frontender = new Frontender();

frontender.printProfession.call(backender); *// "I am a Frontend Developer"*

frontender.printProfession.apply(backender); *// "I am a Frontend Developer"*

const boundPrintProfession = frontender.printProfession.bind(backender);

boundPrintProfession(); *// "I am a Frontend Developer"*

Поэтому методы call, apply или bind со стрелочными функциями не работают.

# Вызов с оператором new

Стрелочные функции нельзя вызвать как конструктор — это приведёт к ошибке:

Скопировать кодJAVASCRIPT

const arrowFunction = () => {

console.log(this);

};

const obj = new arrowFunction(); *// TypeError: arrowFunction is not a constructor*

Функция-конструктор устанавливает в this созданный объект. Но у стрелочной функции своего this быть не может, поэтому вызвать с new её нельзя.

# Когда использовать

Если в теле функции не используется ключевое слово this, можно не задумываться, делать её стрелочной или нет — делайте, как хотите. Например, удобно использовать стрелочные функции в колбэке, потому что это короче:

Скопировать кодJAVASCRIPT

const numbers = [2, 3, 5];

const doubledNumbers = numbers.map(number => number \* 2);

console.log(doubledNumbers); *// 4, 6, 10*

Но если this внутри функции нужен — стоит задуматься, должна она быть стрелочной или нет.

# Внутри функции-конструктора

Если объявить стрелочную функцию внутри функции-конструктора, её значением this навсегда станет функция-конструктор:

Скопировать кодJAVASCRIPT

function Table() {

this.rows = 4;

this.columns = 3;

this.printInfo = () => {

console.log('Table has ' + this.rows + ' rows and ' + this.columns + ' columns');

};

}

const myTable = new Table();

myTable.printInfo(); *// Table has 4 rows and 3 columns*

Стрелочная функция printInfo возьмёт this из замыкания, то есть из функции Table. Так она навсегда привяжет свой this к функции Table. Даже если обратиться к этой функции не как к методу объекта, а отдельно записать в переменную и затем вызвать, она не потеряет контекст:

Скопировать кодJAVASCRIPT

function Table() {

this.rows = 4;

this.columns = 3;

this.printInfo = () => {

console.log('Table has ' + this.rows + ' rows and ' + this.columns + ' columns');

};

}

const myTable = new Table();

const myFunc = myTable.printInfo;

myFunc(); *// Table has 4 rows and 3 columns — this навсегда привязан к Table*

# Важно

Если методы находятся в прототипе, не пользуйтесь стрелочными функциями:

Скопировать кодJAVASCRIPT

function Table() {

this.rows = 4;

this.columns = 3;

}

Table.prototype.printInfo = () => {

console.log('Table has ' + this.rows + ' rows and ' + this.columns + ' columns');

};

const myTable = new Table();

myTable.printInfo(); *// Table has undefined rows and undefined columns*

Значением this метода printInfo станет навсегда равен this глобальной области видимости — то есть window.

**Всё о this. Заключение**

Можно выдохнуть — одна из самых запутанных тем курса позади. Чтобы лучше осознать произошедшее, повторим, что только что было.

На протяжении двух тем вы изучали точки вызова функций. Вы узнали, что привязка this подчиняется четырём правилам. Они действуют от большего приоритета к меньшему:

1. Если функция вызвана с оператором new, используется только что созданный объект.
2. Если функция вызвана одним из методов call, apply или после привязки методом bind, в this оказывается переданный в метод объект.
3. Если вызван метод объекта, тогда this — это объект, на котором этот метод вызван
4. Если ни одно правило не сработало, то идёт в ход привязка по умолчанию: в строгом режиме — undefined, иначе — глобальный объект window.

Есть пятый способ привязки this — у стрелочных функций. Он кардинально отличается от первых четырёх, потому что в этом случае this привязывается во время объявления функции, а не во время вызова.

Теперь вы узнали о всех тайнах this на втором месяце изучения JavaScript. Это ещё далеко не мастерство, но этап знакомства пройден. Даже если вы поняли 10% изложенного, это было не зря, потому что сильно поможет вам развиваться как js-разработчику.