

🏠 → [Язык JavaScript](#) → [Объекты: основы](#)

📅 4 сентября 2022 г.

# Методы объекта, "this"

Объекты обычно создаются, чтобы представлять сущности реального мира, будь то пользователи, заказы и так далее:

```
1 // Объект пользователя
2 let user = {
3   name: "John",
4   age: 30
5 };
```

И так же, как и в реальном мире, пользователь может *совершать действия*: выбирать что-то из корзины покупок, авторизовываться, выходить из системы, оплачивать и т.п.

Такие действия в JavaScript представлены функциями в свойствах.

## Примеры методов

Для начала давайте научим нашего пользователя `user` здороваться:

```
1 let user = {
2   name: "John",
3   age: 30
4 };
5
6 user.sayHi = function() {
7   alert("Привет!");
8 };
9
10 user.sayHi(); // Привет!
```



Здесь мы просто использовали Function Expression (функциональное выражение), чтобы создать функцию приветствия, и присвоили её свойству `user.sayHi` нашего объекта.

Затем мы можем вызвать её как `user.sayHi()`. Теперь пользователь может говорить!

Функцию, которая является свойством объекта, называют *методом* этого объекта.

Итак, мы получили метод `sayHi` объекта `user`.

Конечно, мы могли бы использовать заранее объявленную функцию в качестве метода, вот так:

```
1 let user = {
2   // ...
```



```

3  };
4
5  // сначала, объявляем
6  function sayHi() {
7      alert("Привет!");
8  }
9
10 // затем добавляем в качестве метода
11 user.sayHi = sayHi;
12
13 user.sayHi(); // Привет!

```

### **i** Объектно-ориентированное программирование

Когда мы пишем наш код, используя объекты для представления сущностей реального мира, – это называется **объектно-ориентированным программированием** или сокращённо: «ООП».

ООП является большой предметной областью и интересной наукой самой по себе. Как выбрать правильные сущности? Как организовать взаимодействие между ними? Это – создание архитектуры, и на эту тему есть отличные книги, такие как «Приёмы объектно-ориентированного проектирования. Паттерны проектирования» авторов Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес или «Объектно-ориентированный анализ и проектирование с примерами приложений» Гради Буча, а также ещё множество других книг.

## Сокращённая запись метода

Существует более короткий синтаксис для методов в литерале объекта:

```

1  // эти объекты делают одно и то же
2
3  user = {
4      sayHi: function() {
5          alert("Привет");
6      }
7  };
8
9  // сокращённая запись выглядит лучше, не так ли?
10 user = {
11     sayHi() { // то же самое, что и "sayHi: function(){...}"
12         alert("Привет");
13     }
14 };

```

Как было показано, мы можем пропустить ключевое слово `"function"` и просто написать `sayHi()`.

Нужно отметить, что эти две записи не полностью эквивалентны. Есть тонкие различия, связанные с наследованием объектов (что будет рассмотрено позже), но на данном этапе изучения это неважно. Почти во всех случаях сокращённый синтаксис предпочтителен.

## Ключевое слово «this» в методах

Как правило, методу объекта обычно требуется доступ к информации, хранящейся в объекте, для выполнения своей работы.

Например, коду внутри `user.sayHi()` может потребоваться имя пользователя, которое хранится в объекте `user`.

**Для доступа к информации внутри объекта метод может использовать ключевое слово `this`.**

Значение `this` – это объект «перед точкой», который используется для вызова метода.

Например:



```
1 let user = {
2   name: "John",
3   age: 30,
4
5   sayHi() {
6     // "this" - это "текущий объект".
7     alert(this.name);
8   }
9
10 };
11
12 user.sayHi(); // John
```

Здесь во время выполнения кода `user.sayHi()` значением `this` будет являться `user` (ссылка на объект `user`).

Технически также возможно получить доступ к объекту без ключевого слова `this`, обратившись к нему через внешнюю переменную (в которой хранится ссылка на этот объект):

```
1 let user = {
2   name: "John",
3   age: 30,
4
5   sayHi() {
6     alert(user.name); // "user" вместо "this"
7   }
8
9 };
```

...Но такой код ненадёжен. Если мы решим скопировать ссылку на объект `user` в другую переменную, например, `admin = user`, и перезапишем переменную `user` чем-то другим, тогда будет осуществлён доступ к неправильному объекту при вызове метода из `admin`.

Это показано ниже:



```
1 let user = {
2   name: "John",
3   age: 30,
4
5   sayHi() {
```

```
6     alert( user.name ); // приведёт к ошибке
7   }
8
9 };
10
11
12 let admin = user;
13 user = null; // перезапишем переменную для наглядности, теперь она не хранит с
14
15 admin.sayHi(); // TypeError: Cannot read property 'name' of null
```

Если бы мы использовали `this.name` вместо `user.name` внутри `alert`, тогда этот код бы сработал.

## «this» не является фиксированным

В JavaScript ключевое слово «this» ведёт себя иначе, чем в большинстве других языков программирования. Его можно использовать в любой функции, даже если это не метод объекта.

В следующем примере нет синтаксической ошибки:

```
1 function sayHi() {
2   alert( this.name );
3 }
```

Значение `this` вычисляется во время выполнения кода, в зависимости от контекста.

Например, здесь одна и та же функция назначена двум разным объектам и имеет различное значение «this» в вызовах:

```
1 let user = { name: "John" };
2 let admin = { name: "Admin" };
3
4 function sayHi() {
5   alert( this.name );
6 }
7
8 // используем одну и ту же функцию в двух объектах
9 user.f = sayHi;
10 admin.f = sayHi;
11
12 // эти вызовы имеют разное значение this
13 // "this" внутри функции - это объект "перед точкой"
14 user.f(); // John (this == user)
15 admin.f(); // Admin (this == admin)
16
17 admin['f'](); // Admin (нет разницы между использованием точки или квадратных
```

Правило простое: если вызывается `obj.f()`, то во время вызова `f`, `this` – это `obj`. Так что, в приведённом выше примере это либо `user`, либо `admin`.

### **i** Вызов без объекта: `this == undefined`

Мы даже можем вызвать функцию вообще без объекта:

```
1 function sayHi() {
2   alert(this);
3 }
4
5 sayHi(); // undefined
```



В строгом режиме (`"use strict"`) в таком коде значением `this` будет являться `undefined`. Если мы попытаемся получить доступ к `this.name` – это вызовет ошибку.

В нестрогом режиме значением `this` в таком случае будет *глобальный объект* (`window` в браузере, мы вернёмся к этому позже в главе [Глобальный объект](#)). Это – исторически сложившееся поведение `this`, которое исправляется использованием строгого режима (`"use strict"`).

Обычно подобный вызов является ошибкой программирования. Если внутри функции используется `this`, тогда она ожидает, что будет вызвана в контексте какого-либо объекта.

### **i** Последствия свободного `this`

Если вы до этого изучали другие языки программирования, то вы, вероятно, привыкли к идее «фиксированного `this`» – когда методы, определённые в объекте, всегда имеют `this`, ссылающееся на этот объект.

В JavaScript `this` является «свободным», его значение вычисляется в момент вызова метода и не зависит от того, где этот метод был объявлен, а скорее от того, какой объект вызывает метод (какой объект стоит «перед точкой»).

Эта концепция вычисления `this` в момент исполнения имеет как свои плюсы, так и минусы. С одной стороны, функция может быть повторно использована в качестве метода у различных объектов (что повышает гибкость). С другой стороны, большая гибкость увеличивает вероятность ошибок.

Здесь наша позиция заключается не в том, чтобы судить, является ли это архитектурное решение в языке хорошим или плохим. Скоро мы поймем, как с этим работать, как получить выгоду и избежать проблем.

## У стрелочных функций нет «this»

Стрелочные функции особенные: у них нет своего «собственного» `this`. Если мы ссылаемся на `this` внутри такой функции, то оно берётся из внешней «нормальной» функции.

Например, здесь `arrow()` использует значение `this` из внешнего метода `user.sayHi()`:

```
1 let user = {
2   firstName: "Ilya",
3   sayHi() {
4     let arrow = () => alert(this.firstName);
5     arrow();
6   }
7 };
8
9 user.sayHi(); // Ilya
```



Это особенность стрелочных функций. Она полезна, когда мы на самом деле не хотим иметь отдельное `this`, а скорее хотим взять его из внешнего контекста. Позже в главе [Повторяем стрелочные функции](#) мы увидим больше примеров на эту тему.

## Итого

- Функции, которые находятся в свойствах объекта, называются «методами».
- Методы позволяют объектам «действовать»: `object.doSomething()`.
- Методы могут ссылаться на объект через `this`.

Значение `this` определяется во время исполнения кода.

- При объявлении любой функции в ней можно использовать `this`, но этот `this` не имеет значения до тех пор, пока функция не будет вызвана.
- Функция может быть скопирована между объектами (из одного объекта в другой).
- Когда функция вызывается синтаксисом «метода» — `object.method()`, значением `this` во время вызова является `object`.

Также ещё раз заметим, что стрелочные функции являются особенными — у них нет `this`. Когда внутри стрелочной функции обращаются к `this`, то его значение берётся извне.

## ✓ Задачи

### Использование "this" в литерале объекта

важность: 5

Здесь функция `makeUser` возвращает объект.

Каким будет результат при обращении к свойству объекта `ref`? Почему?

```
1 function makeUser() {
2   return {
3     name: "John",
4     ref: this
5   };
6 }
7
8 let user = makeUser();
9
10 alert( user.ref.name ); // Каким будет результат?
```

решение

Ответ: ошибка.

Проверьте:



```

1  function makeUser() {
2      return {
3          name: "John",
4          ref: this
5      };
6  }
7
8  let user = makeUser();
9
10 alert( user.ref.name ); // Error: Cannot read property 'name' of undef.

```

Это потому, что правила, которые определяют значение `this`, никак не смотрят на объявление объекта. Важен лишь момент вызова.

Здесь значение `this` внутри `makeUser()` равно `undefined`, потому что оно вызывается как функция, а не через «точечный» синтаксис как метод.

Значение `this` одно для всей функции, блоки кода и объектные литералы на него не влияют.

Таким образом, `ref: this` фактически принимает текущее `this` функции `makeUser()`.

Мы можем переписать функцию и вернуть то же самое `this` со значением `undefined`:

```

1  function makeUser(){
2      return this; // на этот раз нет литерала объекта
3  }
4
5  alert( makeUser().name ); // Error: Cannot read property 'name' of unde

```

Как вы можете видеть, результат `alert( makeUser().name )` совпадает с результатом `alert( user.ref.name )` из предыдущего примера.

Вот противоположный случай:

```

1  function makeUser() {
2      return {
3          name: "John",
4          ref() {
5              return this;
6          }
7      };
8  }
9
10 let user = makeUser();
11
12 alert( user.ref().name ); // John

```

Теперь это работает, поскольку `user.ref()` – это метод. И значением `this` становится объект перед точкой `.`.

## Создайте калькулятор

важность: 5

Создайте объект `calculator` (калькулятор) с тремя методами:




- `read()` (читать) запрашивает два значения и сохраняет их как свойства объекта.
- `sum()` (суммировать) возвращает сумму сохранённых значений.
- `mul()` (умножить) перемножает сохранённые значения и возвращает результат.

```
1 let calculator = {  
2   // ... ваш код ...  
3 };  
4  
5 calculator.read();  
6 alert( calculator.sum() );  
7 alert( calculator.mul() );
```

[Запустить демо](#)

[Открыть песочницу с тестами для задачи.](#)

решение



```
1 let calculator = {  
2   sum() {  
3     return this.a + this.b;  
4   },  
5  
6   mul() {  
7     return this.a * this.b;  
8   },  
9  
10  read() {  
11    this.a = +prompt('a?', 0);  
12    this.b = +prompt('b?', 0);  
13  }  
14 };  
15  
16 calculator.read();  
17 alert( calculator.sum() );  
18 alert( calculator.mul() );
```

[Открыть решение с тестами в песочнице.](#)

## Цепь вызовов

важность: 2



У нас есть объект `ladder` (лестница), который позволяет подниматься и спускаться:

```
1  let ladder = {
2    step: 0,
3    up() {
4      this.step++;
5    },
6    down() {
7      this.step--;
8    },
9    showStep: function() { // показывает текущую ступеньку
10     alert( this.step );
11   }
12 };
```

Теперь, если нам нужно выполнить несколько последовательных вызовов, мы можем сделать это так:

```
1  ladder.up();
2  ladder.up();
3  ladder.down();
4  ladder.showStep(); // 1
5  ladder.down();
6  ladder.showStep(); // 0
```

Измените код методов `up`, `down` и `showStep` таким образом, чтобы их вызов можно было сделать по цепочке, например так:

```
1  ladder.up().up().down().showStep().down().showStep(); // показывает 1 затем 0
```

Такой подход широко используется в библиотеках JavaScript.

[Открыть песочницу с тестами для задачи.](#)

решение

Решение состоит в том, чтобы возвращать сам объект из каждого вызова.

```
1  let ladder = {
2    step: 0,
3    up() {
4      this.step++;
5      return this;
6    },
7    down() {
8      this.step--;
9      return this;
10   },
11   showStep() {
12     alert( this.step );
13     return this;
14   }
15 };
```

```
14     }  
15   };  
16  
17   ladder.up().up().down().showStep().down().showStep(); // показывает 1 ;
```

Мы также можем записать один вызов на одной строке. Для длинных цепей вызовов это более читабельно:

```
1  ladder  
2    .up()  
3    .up()  
4    .down()  
5    .showStep() // 1  
6    .down()  
7    .showStep(); // 0
```

[Открыть решение с тестами в песочнице.](#)



[Предыдущий урок](#)

[Следующий урок](#)



Поделиться



[Карта учебника](#)

Проводим [курсы по JavaScript и фреймворкам.](#)



## 💬 Комментарии

- Если вам кажется, что в статье что-то не так - вместо комментария напишите [на GitHub](#).
- Для одной строки кода используйте тег `<code>`, для нескольких строк кода — тег `<pre>`, если больше 10 строк — ссылку на песочницу ([plnkr](#), [JSBin](#), [codepen...](#))
- Если что-то непонятно в статье — пишите, что именно и с какого места.

Присоединиться к обсуждению...

ВОЙТИ С ПОМОЩЬЮ

ИЛИ ЧЕРЕЗ DISQUS ?

Имя



65

Поделиться

Лучшие

Новые

Старые



**Джун на фронте**

день назад



Приглашаю в **блог JS-прогресса** !

В **тг Джун на фронте** я врываюсь в веб! Учусь делать динамичные сайты, без банальных конструкторов типа Тильды 🙄

Мы вместе пройдем собесы, отправим первый отклик, выполним тестовык, разберем алгоритмы и окунемся в море кода!

0

0

Ответить • Поделиться ›

J

**JuranTouran**

10 дней назад



//Последняя задача

```
let ladder = {
  step: 0,
  up() {
    this.step++;
    return this
  },
  down() {
    this.step--;
    return this
  },
  showStep: function() {
    alert( this.step );
    return this
  }
};
```

0

0

Ответить • Поделиться ›

K

**KingNut Academy**

14 дней назад



Разбор задач, тестов и всего что поможет при прохождении собеседования на должность frontend developer (html, css, js, ts, react).

Ссылка на TF Interview masters

— — — — —

Пожалуйста объясните мне последнюю задачу. В предыдущих главах проходили, что если плюс стоит после, `this.step++`, то возвращается 0. Т.е без прибавления 1. Тогда почему возвращается вот так:

```
ladder.up(); // 0
ladder.up(); // 1
ladder.down(); // 2
ladder.showStep(); // 1
ladder.down();
ladder.showStep(); // 0
```

Я добавила alert перед каждый `this.step`, чтобы увидеть что показывается

0 0 Ответить • Поделиться ›



**Player\_50B** → Katya Klep

месяц назад



"Префиксная форма" возвращает сразу новое значения.

```
let a = 1;
console.log(++a); // 2
console.log(a); // 2
```

0 0 Ответить • Поделиться ›



**Player\_50B** → Katya Klep

месяц назад



Так все работает как нужно, это называется "постфиксная форма", она возвращает старое значения, но при этом прибавляет его.

```
let a = 1;
console.log(a++); // 1
console.log(a); // 2
```

0 0 Ответить • Поделиться ›



**Katya Klep** → Player\_50B

месяц назад



Но ведь тогда `Ladder.down` не должно быть 2? И даже если оно 2, `Ladder.showStep` показывает 1, если оно показывает положение вещей, почему т.е 2. ?

0 0 Ответить • Поделиться ›



**eternal elers**

→ Katva Klen





**eternal\_elers** · 1 мая 2019  
месяц назад · edited

Если я правильно понял ваш код, то он примерно такой:

```
let ladder = {
  step: 0,
  up() {
    console.log(this.step);
    this.step++;
    return this;
  },
  down() {
    console.log(this.step);
    this.step--;
    return this;
  },
  showStep() {
    console.log(this.step);
    return this;
  },
};
```

[показать больше](#)

2 0 Ответить · Поделиться ›



**Katya Klep**  
месяц назад

→ eternal\_elers



Но почему ladder.step становится 2? Если при втором вызове ladder.up стало 1, то следующий вызов ladder.down также должен сначала показать старое значение т.е 1?

0 0 Ответить · Поделиться ›



**Наталья**

25 дней назад

→ Katya Klep



Тут надо понять тонкость. И это пытались объяснить. Например let step = 0. После оператора step++ у нас получается два значения (0 и 1). Потому, что это постфиксная операция. И, если вместо step++ у нас стоял бы alert(step++), т.е. сразу был бы вывод без перехода к новому оператору, вывелось бы старое значение (0). Но, если после первого step++ идет переход к следующему оператору (напр. step--), то во время перехода старое значение исчезает и работа идет уже с новым (1). В результате действия второго оператора получаем два новых значения. В нашем случае (1 и 0). Следующий переход к новому оператору (напр. alert(step)). Опять при переходе исчезает старое значение (1) и в результате выводится (0).

0 0 Ответить · Поделиться ›



**Katya Klep**  
20 дней назад

→ Наталья



Так второй ladder.up показывает 1. Почему же следующий

ladder.down показывает 2? Откуда оно взялось?

0 0 Ответить • Поделиться ›

Д

Давид

→ Katya Klep

месяц назад

— 🚩

у вас до down два раз было up ответ уже 2, и до того как функция выполняет step-- в функции down вы добавили alert до действия и поэтому показывает 2.

0 0 Ответить • Поделиться ›

К

Katya Klep

→ Давид

20 дней назад

— 🚩

Два раза up, но ведь при первом вызове покажет 0(старое значение) а при втором 1.

0 0 Ответить • Поделиться ›

С

Sultan Agdarbekov

→ Katya Klep

19 дней назад edited

— 🚩

Можете показать код который вы тестите и с которым у вас вопрос , Вы наверно выводите alert до того как произойдет step-- , и по этому у вас 2 а не 1

0 0 Ответить • Поделиться ›

К

Katya Klep

→ Sultan Agdarbekov

19 дней назад

— 🚩

Но ведь на втором вызове up у нас 1 получается

0 0 Ответить • Поделиться ›

К

Katya Klep

→ Sultan Agdarbekov

19 дней назад

— 🚩

Вот мой код:

```
let ladder = {
  step: 0,
  up() {
    alert( this.step++);
  },
  down() {
    alert (this.step--);
  },
  showStep: function() { // показывает текущую ступеньку
    alert( this.step );
  }
};
ladder.up(); // 0
ladder.up();// 1
ladder.down(); // 2
ladder.showStep(); // 1
```

```
ladder.down();
ladder.showStep(); // 0
```

0 0 Ответить • Поделиться ›



**Sultan Agdarbekov**

→ Katya Klep



19 дней назад edited

```
let ladder = {
  step: 0,
  up() {
    this.step++
    console.log(this.step)
  },
  down() {
    this.step--;
    console.log(this.step)
  },
  showStep: function() {
    console.log(this.step);
  }
};
ladder.up(); // 1
ladder.up(); // 2
ladder.down(); // 1
ladder.showStep(); // 1
```

[показать больше](#)

0 0 Ответить • Поделиться ›



**JabaDushnila**

2 месяца назад edited



Объяснение решения для №1 непонятно вообще что объясняет

0 0 Ответить • Поделиться ›



**Player\_50B**

→ JabaDushnila



месяц назад

Грубо говоря мы возвращаем user.[Window].name, а не сам объект.

1 0 Ответить • Поделиться ›



**Player\_50B**

→ JabaDushnila



месяц назад edited

"this" ссылается на глобальный контекст, то есть на "Window", если бы мы создали метод, тогда он ссылался бы на контекст объекта:

```
function makeUser() {
  return {
    name: "John",
    ref: function () {
      return this;
    }
  };
}

let user = makeUser();
```

```
let user = makeUser( //  
alert( user.ref().name ); // John
```

1 0 Ответить • Поделиться ›

Загрузить ещё комментарии

Подписаться Privacy

Не продавайте мои данные