# Методы объекта, "this"

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

```
// Объект пользователя
let user = {
  name: "Джон",
  age: 30
};
```

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 свойствами-функциями объекта.

## Примеры методов

Для начала давайте научим нашего пользователя `user` здороваться:

```
let user = {
  name: "Джон",
  age: 30
};

user.sayHi = function() {
  alert("Привет!");
};

user.sayHi(); // Привет!
```

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

Затем мы вызвали её. Теперь пользователь может говорить!

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

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

Конечно, мы могли бы заранее объявить функцию и использовать её в качестве метода, примерно так:

```
let user = {
  // ...
};

// сначала объявляем
function sayHi() {
  alert("Привет!");
};

// затем добавляем в качестве метода
user.sayHi = sayHi;

user.sayHi(); // Привет!
```

**Объектно-ориентированное программирование**

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

ООП является большой предметной областью и интересной наукой само по себе. Как выбрать правильные сущности? Как организовать взаимодействие между ними? Это – создание архитектуры, и есть хорошие книги по этой теме, 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:

```
// эти объекты делают одно и то же (одинаковые методы)

user = {
  sayHi: function() {
    alert("Привет");
  }
};

// сокращённая запись выглядит лучше, не так ли?
user = {
  sayHi() { // то же самое, что и "sayHi: function()"
    alert("Привет");
  }
};
```

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. В большинстве случаев сокращённый синтаксис предпочтителен.

## Ключевое слово «this» в методах

Как правило, методу объекта необходим доступ к информации, которая хранится в объекте, чтобы выполнить с ней какие-либо действия (в соответствии с назначением метода).

Например, коду внутри `user.sayHi()` может понадобиться 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 использовался для вызова метода.

Например:

```
let user = {
  name: "Джон",
  age: 30,

  sayHi() {
    // this - это "текущий объект"
    alert(this.name);
  }

```



```
10  };
11
12  user.sayHi(); // Джон
```

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`, ссылаясь на него через внешнюю переменную (в которой хранится ссылка на этот объект):

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

...Но такой код будет ненадёжным. 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: "Джон",
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; // обнулим переменную для наглядности, теп
14
15 admin.sayHi(); // Ошибка! Внутри sayHi() используется u
```

Если мы используем `this.name` вместо `user.name` внутри `alert`, тогда этот код будет работать.

## «this» не является фиксированным

В JavaScript ключевое слово «this» ведёт себя иначе, чем в большинстве других языков программирования. Оно может использоваться в любой функции.

В этом коде нет синтаксической ошибки:

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

Значение `this` вычисляется во время выполнения кода и зависит от контекста.

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

```
1  let user = { name: "Джон" };
```



```
let admin = { name: "Админ" };

function sayHi() {
  alert( this.name );
}

// используем одну и ту же функцию в двух объектах
user.f = sayHi;
admin.f = sayHi;

// вызовы функции, приведённые ниже, имеют разное значе
// "this" внутри функции является ссылкой на объект, ко
user.f(); // Джон  (this == user)
admin.f(); // Админ  (this == admin)

admin['f'](); // Админ (неважен способ доступа к методу
```

Правило простое: при вызове `obj.f()` значение `this` внутри `f` равно `obj`. Так что, в приведённом примере это `user` или `admin`.

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

Мы даже можем вызвать функцию вовсе без использования объекта:

```
function sayHi() {
  alert(this);
}

sayHi(); // undefined
```

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

В нестрогом режиме значением `this` в таком случае будет *глобальный объект* (`window` для браузера, 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 ожидается, что она будет вызываться в контексте какого-либо объекта.

**Последствия свободного `this`**

Если вы до этого изучали другие языки программирования, тогда вы, скорее всего, привыкли к идее "фиксированного `this`" – когда методы, определённые внутри объекта, всегда сохраняют в качестве значения `this` ссылку на свой объект (в котором был определён метод).

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

Эта идея 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 мы не будем судить о том, является ли это решение в языке хорошим или плохим. Мы должны понимать, как с этим работать, чтобы получать выгоды и избегать проблем.

# Внутренняя реализация: Ссылочный тип

> **Продвинутая возможность языка**
>
> Этот раздел объясняет сложную тему, чтобы лучше понимать некоторые запутанные случаи.
>
> Если вы хотите продвигаться быстрее, его можно пропустить или отложить.

Некоторые хитрые способы вызова метода приводят к потере значения `this`, например:

```
let user = {
  name: "Джон",
  hi() { alert(this.name); },
  bye() { alert("Пока"); }
};

user.hi(); // Джон (простой вызов метода работает хорош

// теперь давайте попробуем вызывать user.hi или user.b
// в зависимости от имени пользователя user.name
(user.name == "Джон" ? user.hi : user.bye)(); // Ошибка
```

В последней строчке кода используется условный оператор `?`, который определяет, какой будет вызван метод (`user.hi` или `user.bye`) в зависимости от выполнения условия. В данном случае будет выбран `user.hi`.

Затем метод тут же вызывается с помощью скобок `()`. Но вызов не работает как положено!

Вы можете видеть, что при вызове будет ошибка, потому что значением `"this"` внутри функции становится `undefined` (полагаем, что у нас строгий режим).

Так работает (доступ к методу объекта через точку):

```
user.hi();
```

Так уже не работает (вызываемый метод вычисляется):

```
(user.name == "Джон" ? user.hi : user.bye)(); // Ошибка
```

Почему? Если мы хотим понять, почему так происходит, давайте разберёмся (заглянем под капот), как работает вызов методов (`obj.method()`).

Присмотревшись поближе, в выражении `obj.method()` можно заметить две операции:

1. Сначала оператор точка `'.'` возвращает свойство объекта – его метод (`obj.method`).
2. Затем скобки `()` вызывают этот метод (исполняется код метода).

Итак, каким же образом информация о `this` передаётся из первой части во вторую?

Если мы поместим эти операции в отдельные строки, то значение `this`, естественно, будет потеряно:

```
let user = {
  name: "Джон",
  hi() { alert(this.name); }
};

```



```
6
7  // разделим получение метода объекта и его вызов в разны
8  let hi = user.hi;
   hi(); // Ошибка, потому что значением this является und
```

Здесь `hi = user.hi` сохраняет функцию в переменной, и далее в последней строке она вызывается полностью сама по себе, без объекта, так что нет `this`.

**Для работы вызовов типа `user.hi()`, JavaScript использует трюк – точка `'.'` возвращает не саму функцию, а специальное значение «ссылочного типа», называемого Reference Type.**

Этот ссылочный тип (Reference Type) является внутренним типом. Мы не можем явно использовать его, но он используется внутри языка.

Значение ссылочного типа – это «триплет»: комбинация из трёх значений `(base, name, strict)`, где:

- `base` – это объект.
- `name` – это имя свойства объекта.
- `strict` – это режим исполнения. Является true, если действует строгий режим (`use strict`).

Результатом доступа к свойству `user.hi` является не функция, а значение ссылочного типа. Для `user.hi` в строгом режиме оно будет таким:

```
1  // значение ссылочного типа (Reference Type)
2  (user, "hi", true)
```

Когда скобки `()` применяются к значению ссылочного типа (происходит вызов), то они получают полную информацию об объекте и его методе, и могут поставить правильный `this` (`=user` в данном случае, по `base`).

Ссылочный тип – исключительно внутренний, промежуточный, используемый, чтобы передать информацию от точки `.` до вызывающих скобок `()`.

При любой другой операции, например, присваивании `hi = user.hi`, ссылочный тип заменяется на собственно значение `user.hi` (функцию), и дальше работа уже идёт только с ней. Поэтому дальнейший вызов происходит уже без `this`.

Таким образом, значение `this` передаётся правильно, только если функция вызывается напрямую с использованием синтаксиса точки `obj.method()` или квадратных скобок `obj['method']()` (они делают то же самое). Позднее в этом учебнике мы изучим различные варианты решения проблемы потери значения `this`. Например, такие как func.bind().

## У стрелочных функций нет «this»

Стрелочные функции особенные: у них нет своего «собственного» `this`. Если мы используем `this` внутри стрелочной функции, то его значение берётся из внешней «нормальной» функции.

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

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

Это является особенностью стрелочных функций. Они полезны, когда мы на самом деле не хотим иметь отдельное значение `this`, а хотим брать его из

внешнего контекста. Позднее в главе Повторяем стрелочные функции мы увидим больше примеров на эту тему.

## Итого

- Функции, которые находятся в объекте в качестве его свойств, 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.
- Эта функция может быть скопирована между объектами (из одного объекта в другой).
- Когда функция вызывается синтаксисом «метода» – `object.method()`, значением `this` во время вызова является объект перед точкой.

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

## Задачи

### Проверка синтаксиса

важность: 2

Каким будет результат выполнения этого кода?

```
let user = {
  name: "Джон",
  go: function() { alert(this.name) }
}

(user.go)()
```

P.S. Здесь есть подвох :)

решение

### Объясните значение "this"

важность: 3

В представленном ниже коде мы намерены вызвать `obj.go()` метод 4 раза подряд.

Но вызовы `(1)` и `(2)` работают иначе, чем `(3)` и `(4)`. Почему?

```
let obj, method;

obj = {
  go: function() { alert(this); }
};

obj.go();               // (1) [object Object]

(obj.go)();             // (2) [object Object]

(method = obj.go)();    // (3) undefined

(obj.go || obj.stop)(); // (4) undefined
```

решение



### Использование "this" в литерале объекта

важность: 5

Здесь функция `makeUser` возвращает объект.

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

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

let user = makeUser();

alert( user.ref.name ); // Каким будет результат?
```

решение

### 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.

```
let calculator = {
  // ... ваш код ...
};

calculator.read();
alert( calculator.sum() );
alert( calculator.mul() );
```

Запустить демо

Открыть песочницу с тестами для задачи.

решение

### Цепь вызовов

важность: 2

Это `ladder` (лестница) – объект, который позволяет подниматься вверх и спускаться:

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

Поделиться

Редактировать на GitHub

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

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

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

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

Такой подход широко используется в библиотеках JavaScript.

Открыть песочницу с тестами для задачи.

решение



## Комментарии

перед тем как писать...