GeekBrains

Основы ООП. Контекст this

Курс JavaScript про ECMAScript

Оглавление

Объекты и их методы

Программисты давно поняли, что проще всего создавать алгоритмы, оперируя в них объектами, которые отображают реальный мир. Так практически любую сущность реального мира можно представить в виде объекта с некоторыми свойствами и методами.

Возьмем к примеру робот-пылесос, у него могут быть разные характеристики, такие как мощность двигателя, емкость аккумулятора, время работы без подзарядки, объем контейнера под мусор, также у него есть разные датчики, которые могут быть представлены булевыми переменными, например заполнен ли контейнер для мусора, или датчик препятствия. А также у робота-пылесоса есть методы, которые делают его не просто набором свойств, но и добавляют функциональности: самая главная - уборка комнаты, и дополнительные, отправиться на подзарядку, когда заряд на исходе, или активировать дезинфекцию ультрафиолетовой лампой. Все это легко можно запрограммировать в виде объекта, давайте напишем такой объект:

Листинг 1.

```
// Объект робот-пылесос.
const Roomba = { // Есть негласное правило называть объекты в
алгоритмах с большой буквы.
   // Обычно сначала объявляют свойства объекта.
   model: "Romba-1",
   power: 200,
   batterySize: 2100,
   boxSize: 0.5,
   workTime: 45,
   counterOfStarts: 0,
   isFull: false,
   isObstacle: false,
   isUVLampOn: false,

   // После свойств объявляют его методы.
   startCleaning: function () {
       this.counterOfStarts++;
       console.log('I am cleaning... I have been started: ',
this.counterOfStarts, 'times.');
   },
   goCharge: function () {
       console.log('I am going to charge...');
   },
   switchUVLamp: function () {
       this.isUVLampOn = !this.isUVLampOn;
       console.log(`UV lamp is ${this.isUVLampOn ? 'working' :
'not working'}.`);
   }
};
```

В этом уроке мы подробно рассмотрим как взаимодействовать с методами объектов, а о самих объектах и объектно-ориентированном программировании мы поговорим в следующем уроке.

У нашего объекта получилось три метода - это:

- **`startCleaning`** - при вызове этого метода объект увеличивает счетчик стартов на единицу и выводится сообщение в консоль, что пылесос принялся за уборку и сколько раз он уже был запущен.
- **`goCharge`** - отправиться на зарядку, при его вызове в консоль выводится сообщение что пылесос отправился заряжаться.

- **`switchUVLamp`** - переключить состояние ультрафиолетовой лампы. Стандартный toggle переключать (вкл/выкл) и сообщение в консоли о работе лампы

Давайте посмотрим как мы можем взаимодействовать с объектом:

Листинг 2.

```
// Обращение к свойствам объекта.
console.log(Roomba.model); // "Romba-1"
console.log(Roomba.isFull); // false

// Вызов методов объекта.
Roomba.startCleaning(); // 'I am cleaning... I have been started:
1 times.'

// Установим свойства объекта isUVLampOn в true, чтобы
продемонстрировать результат работы метода switchUVLamp.
Roomba.isUVLampOn = true;

// Результат вызова следующего метода зависит от значения,
хранящегося в свойстве объекта, а также от того как этот метод
был вызван (об этом поговорим чуть ниже).
Roomba.switchUVLamp(); // 'UV lamp is not working.'

Roomba.goCharge(); // 'I am going to charge...'
```

Все достаточно просто и тривиально, мы обращаемся к свойствам объекта через точку, точно также через точку можем вызывать его методы. В методах `startCleaning` и `switchUVLamp` вы можете видеть обращение к свойствам объектов через ключевое слово `this`. Т.к. объект это не функция, то к объявленным в нем свойствам и методам нельзя обратиться из методов этого объекта как к глобальным переменным. Вот мы и подошли к одному из самых частых вопросов на собеседовании, конечно же к контексту (this). Давайте более детально его рассмотрим.

this

this - это ключевое слово в языке JavaScript, которое позволяет обратиться к свойствам и методам объекта внутри его методов. А также ключевое слово **this**

доступно в любой функции, и либо принимает значение объекта, который являлся контекстом при вызове функции, либо undefined.

Давайте разбираться подробнее что такое контекст внутри функции, откуда он берется, и как понять какой контекст будет в нашей функции, и как его переопределить. Для начала попробуем объявить простую функцию и посмотреть что у неё лежит в this, если она не объявлена в объекте.

Листинг 3.

```
// Работа с this
const checkThis = function() {
 console.log(this);
}

checkThis(); // Window {0: global, window: Window, self: Window,
document: document, name: "", location: Location, …}
```

Как мы можем видеть, если мы объявляем функцию вне объекта, в её **this** лежит глобальный объект **Window** (если запускать в браузере, если это будет node.js то будет объект global). Функция, объявленная вне какого-либо пользовательского объекта является методом глобального объекта, поэтому её **this** указывает на глобальный объект. А теперь давайте попробуем создать такую функцию внутри пользовательского объекта:

Листинг 4.

```
const checkThisInObject = {
   testProperty: true,
   checkThis: function () {
       console.log(this);
   },
};

checkThisInObject.checkThis(); // {testProperty: true, checkThis:
ƒ}
```

На этот раз в качестве **this** в функции мы получили наш объект. Вроде все просто, если мы хотим обратиться к свойствам и методам нашего объекта надо просто использовать **this** как ссылку на него, и нам будет все доступно. Но есть некоторые тонкости с этим this, которые мы сейчас рассмотрим.

Одалживание метода

Что если мы хотим одолжить метод одного объекта и использовать его в другом объекте. Для этого давайте создадим еще один объект робота-пылесоса, который будет иметь улучшенные характеристики, но будет иметь такую же функциональность, как первая модель:

Листинг 5.

```
// Объект Roomba - робот-пылесос. Код самого объекта смотри в листинге 1.
// Объект робот-пылесос модель Tango.
const Tango = { // Есть негласное правило называть объекты в алгоритмах с большой буквы.
   // Обычно сначала объявляют свойства объекта.
   model: "Tango-1",
   power: 300,
   batterySize: 3200,
   boxSize: 0.7,
   workTime: 60,
   counterOfStarts: 0,
   isFull: false,
   isObstacle: false,
   isUVLampOn: false,

   // После свойств объявляют его методы. А так как методы у новой модели такие же как и у старой, давайте позаимствуем их у объекта Roomba.
   startCleaning: Roomba.startCleaning,
   goCharge: Roomba.goCharge,
   switchUVLamp: Roomba.switchUVLamp,
};
```

Тут мы воспользовались таким приемом, который называется одалживание метода, мы просто взяли и скопировали методы первого объекта, во второй объект. Давайте попробуем их вызвать и посмотрим что получится:

Листинг 6.

```
// Объект Roomba - робот-пылесос. Код самого объекта смотри в листинге 1.
// Обращение к свойствам и методом объекта Roomba. Код обращения смотри в листинге 2.
// Объект Tango - робот-пылесос. Код самого объекта смотри в листинге 5.

// Обращение к свойствам объекта Tango.
console.log(Tango.model); // "Tango-1"
console.log(Tango.isFull); // false

// Вызов методов объекта.
Tango.startCleaning(); // 'I am cleaning... I have been started: 1 times.'

// Установим свойства объекта isUVLampOn в true, чтобы продемонстрировать результат работы метода switchUVLamp.
Tango.isUVLampOn = true;

// Результат вызова следующего метода зависит от значения, хранящегося в свойстве объекта, а также от того как этот метод был вызван (об этом поговорим чуть ниже).
Tango.switchUVLamp(); // 'UV lamp is not working.'

Tango.goCharge(); // 'I am going to charge...'
```

Все работает, отлично. Давайте создадим третего робота, опять же отличающегося только свойствами, а функциональность будет та же.

Листинг 7.

```
// Объект робот-пылесос модель Samba.
const Samba = {
   model: "Samba-1",
   power: 250,
   batterySize: 2500,
   boxSize: 0.5,
   workTime: 50,
   counterOfStarts: 0,
   isFull: false,
   isObstacle: false,
   isUVLampOn: false,
   // На этот раз мы не будем создавать методы в объекте, мы постараемся их заимствовать непосредственно перед использованием.
};
```

Мы не стали объявлять у этого объекта методы, и попробуем их добавить в объект непосредственно перед вызовом:

Листинг 8.

```
// Объект Samba - робот-пылесос. Код самого объекта смотри в листинге 7.

// Обращение к свойствам объекта Samba.
console.log(Samba.model); // "Samba-1"
console.log(Samba.isFull); // false
// Одолжим методы из объекта Roomba.
Samba.startCleaning = Roomba.startCleaning;
Samba.switchUVLamp = Roomba.switchUVLamp;
Samba.goCharge = Roomba.goCharge;

// Вызов методов объекта.
Samba.startCleaning(); // 'I am cleaning... I have been started: 1 times.'

// Установим свойства объекта isUVLampOn в true, чтобы продемонстрировать результат работы метода switchUVLamp.
Samba.isUVLampOn = true;

// Результат вызова следующего метода зависит от значения, хранящегося в свойстве объекта, а также от того как этот метод был вызван (об этом поговорим чуть ниже).
Samba.switchUVLamp(); // 'UV lamp is not working.'

Samba.goCharge(); // 'I am going to charge...'
```

Супер, так снова всё работает, так что у нас развязаны руки и вы можете протестировать эти 2 вариант самостоятельно, просто для себя создайте такой же пример будет Листинг 9 и 10.

Привязка контекста

Продолжим работу с нашим роботам, и попробуем написать небольшую программу тестирования пылесоса, мы будем через небольшие интервалы отдавать пылесосу команды, и смотреть как они отработали:

Листинг 11.

```
// Объект Roomba - робот-пылесос. Код самого объекта смотри в
листинге 1.

// Обращение к свойствам объекта.
console.log(Roomba.model); // "Romba-1"
console.log(Roomba.isFull); // false

// Вызов методов объекта.
setTimeout(Roomba.startCleaning, 1000);

// Установим свойства объекта isUVLampOn в true, чтобы
продемонстрировать результат работы метода switchUVLamp.
Roomba.isUVLampOn = true;

// Результат вызова следующего метода зависит от значения,
хранящегося в свойстве объекта, а также от того как этот метод
был вызван (об этом поговорим чуть ниже).
setTimeout(Roomba.switchUVLamp, 2000);

setTimeout(Roomba.goCharge, 3000);

// I am cleaning... I have started:  NaN times.
// UV lamp is working.
// I am going to charge...
```

Мы получили немного странный результат. Количество запусков пылесоса стало NaN, а ультрафиолетовая лампа не выключилась. Почему это произошло, давайте разбираться. Когда мы вызывали методы объекта напрямую, после его создания, функция вызывалась имея возможность получить доступ к объекту, но когда функция вызывается внутри метода **setTimeout**, то эта функция теряет доступ к своему объекту, и ключевое слово **this** в такой функции получает значение undefined. Вот тут и вступает в игру **контекст** вызова функции. Каждая функция вызывается в **контексте** некоторого объекта, если эта функция определена вне какого-то пользовательского объекта, то её контекстом будет глобальный объект (например window в браузере), а если определена в пользовательском объекте, и вызвана в нём, то контекстом для неё будет этот пользовательский объект. Когда же мы вызываем функцию в отрыве от её объекта, как это происходит при вызове её из setTimeout, то её контекстом становится undefined. Так происходит потому, что мы одалживаем метод у объекта, и функция setTimeout копирует нашу функцию, для того чтобы вызвать её позже, но когда она вызывается доступа к объекту уже нет. Как мы можем это исправить? Один из вариантов обернуть метод в анонимную функцию, и вызвать в ней, тогда эта анонимная функция в своем лексическом окружении сохранит ссылку на объект, из которого наш метод будет вызываться:

Листинг 12.

```
// Объект Roomba - робот-пылесос. Код самого объекта смотри в
листинге 1.

// Обращение к свойствам объекта.
console.log(Roomba.model); // "Romba-1"
console.log(Roomba.isFull); // false

// Вызов методов объекта.
setTimeout(function () {
   Roomba.startCleaning();
}, 1000);

// Установим свойства объекта isUVLampOn в true, чтобы
продемонстрировать результат работы метода switchUVLamp.
Roomba.isUVLampOn = true;

// Результат вызова следующего метода зависит от значения,
хранящегося в свойстве объекта, а также от того как этот метод
был вызван (об этом поговорим чуть ниже).
setTimeout(function () {
   Roomba.switchUVLamp();
}, 2000);

setTimeout(function () {
   Roomba.goCharge();
}, 3000);

// I am cleaning... I have started:  1 times.
// UV lamp is not working.
// I am going to charge...
```

Сработало, но каждый раз оборачивать метод в анонимную функцию не очень удобно, есть способы лучше. Все они связаны с привязкой контекста (нужного нам объекта) во время вызова функции. Это уже знакомый нам метод **call**, а также еще два метода **apply** и **bind**.

Метод **call** позволяет вызвать функцию и явно указать с каким объектом контекста её выполнить (передать в качестве первого аргумента объект, который будет доступен в функции через ключевое слово **this**). Давайте посмотрим на примере с пылесосом:

Листинг 13.

```
// Объект Roomba - робот-пылесос. Код самого объекта смотри в
листинге 1.

// Обращение к свойствам объекта.
console.log(Roomba.model); // "Romba-1"
console.log(Roomba.isFull); // false

// Вызов методов объекта.
// Вызов метода объекта через call с явной передачей объекта
робота-пылесоса в качестве контекста.
Roomba.startCleaning.call(Roomba); // I am cleaning... I have
started:  1 times.
// Тут этот пример не очень показателен, т.к. Мы и так имели
доступ к объекту, а внутри setTimeout использовать call возможно
только обернув все это в анонимную функцию, но тоже бессмысленно,
потому что тогда мы снова имеем доступ к объекту, как видели в
прошлом примере. Но мы можем передать в call другой объект и
увидеть что функция вызывается в контексте другого объекта:

// Создадим фиктивный объект робота, который содержит только одно
свойство, необходимое для работы функции и сразу же зададим ему
первоначальное значение, отличное от того, которое задано у
робота, для наглядности.
const notARobot = {
   counterOfStarts: 10,
};

Roomba.startCleaning.call(notARobot); // I am cleaning... I have
been started:  11 times.
```

Как мы видим, метод call позволил нам вызвать метод пользовательского объекта, но при этом указать в качестве контекста совсем другой объект и это сработало. Таким способом мы можем использовать метод **call** для вызова любой функции с нужным нам контекстом. Если вызываемая функция принимает аргументы, то их можно указать после объекта контекста, второй и все последующие аргументы метода **call** будут переданы как аргументы вызываемой функции.

Подобно методу **call** можно использовать метод **apply**, который также позволяет вызвать функцию и передать необходимый контекст, единственным отличием от call, метод apply принимает аргументы, которые необходимо передать в вызываемую функцию не списком через запятую, а в виде массива, что порой удобнее. В нашем пример методы не принимают аргументов, но если бы принимали, это могло бы выглядеть вот так:

Листинг 14.

```
// Объект Roomba - робот-пылесос. Код самого объекта смотри в
первом листинге.
const notARobot = {
   counterOfStarts: 10,
};
// Пример использования метода apply, для вызова функции с
передачей в качестве контекста объекта notARobot и передачей в
неё аргументов arg1, arg2, arg3.
Roomba.startCleaning.apply(notARobot, [arg1, arg2, arg3]); // I
am cleaning... I have been started:  11 times.
```

И последний метод для привязки контекста это **bind** (от английского bind - связывать) - это самый часто используемый метод, т.к. позволяет привязать контекст к функции раз и навсегда, и в дальнейшем мы можем просто вызывать функции и быть уверены, что она будет вызвана в контексте нужного нам объекта. Именно он поможет нам, чтобы починить наш алгоритм тестирования робота с использованием setTimeout. Метод работает очень просто, его нужно вызвать для необходимой нам функции и передать в него единственный аргумент - объект в контексте которого мы хотим в дальнейшем вызывать нашу функцию, и наша функция будет привязана к этому контексту навсегда.

Листинг 15.

```
// Объект Roomba - робот-пылесос. Код самого объекта смотри в
листинге 1.
// Обращение к свойствам объекта.
console.log(Roomba.model); // "Romba-1"
console.log(Roomba.isFull); // false

// Вызов методов объекта.
// В setTimeout мы передаем не просто наш метод, а функцию,
которая привязана к нашему объекту. Метод bind возвращает новую
функцию, с уже привязанным контекстом, именно она вызывается по
истечении времени.
setTimeout(Roomba.startCleaning.bind(Roomba), 1000);

// Установим свойства объекта isUVLampOn в true, чтобы
продемонстрировать результат работы метода switchUVLamp.
Roomba.isUVLampOn = true;

// Результат вызова следующего метода зависит от значения,
хранящегося в свойстве объекта, а также от того как этот метод
```

```
был вызван (об этом поговорим чуть ниже).
setTimeout(Roomba.switchUVLamp.bind(Roomba), 2000);
setTimeout(Roomba.goCharge.bind(Roomba), 3000);

// I am cleaning... I have been started:  1 times.
// UV lamp is not working.
// I am going to charge...
```

Все заработало как надо.

Если вы обратили внимание, то все примеры были даны с функциями определенными через ключевое слово function, и нигде не использовалась жирная стрелка (fat arrow) из ES2015, давайте попробуем создать объект, у которого методы будут определены таким образом, и посмотрим что получится:

Листинг 16.

```
const Roomba = {
   model: "Romba-1",
   power: 200,
   batterySize: 2100,
   boxSize: 0.5,
   workTime: 45,
   counterOfStarts: 0,
   isFull: false,
   isObstacle: false,
   isUVLampOn: false,

   // После свойств объявляют его методы.
   startCleaning: () => {
       this.counterOfStarts++;
       console.log('I am cleaning... I have started: ',
this.counterOfStarts, 'times.');
   },
   goCharge: () => {
       console.log('I am going to charge...');
   },
   switchUVLamp: () => {
       this.isUVLampOn = !this.isUVLampOn;
       console.log(`UV lamp is ${this.isUVLampOn ? 'working' :
'not working'}.`);
```

```
    }
};
```

И попробуем вызвать наши методы разными способами:

Листинг 17.

```
// Объект Roomba - робот-пылесос. Код самого объекта смотри в
листинге 16, с использованием синтаксиса ES2015.

// Вызов методов объекта.
Roomba.startCleaning(); // I am cleaning... I have started:  NaN
times.
Roomba.startCleaning.call(Roomba); // I am cleaning... I have
started:  NaN times.
Roomba.startCleaning.apply(Roomba); // I am cleaning... I have
started:  NaN times.
const bindedMethod = Roomba.startCleaning.bind(Roomba);
bindedMethod(); // I am cleaning... I have been started:  NaN
times.

setTimeout(Roomba.startCleaning.bind(Roomba), 1000); // I am
cleaning... I have started:  NaN times.
```

Как мы можем видеть ни один метод не сработал, мы везде получили NaN, почему так? Ответ на этот вопрос кроется в спецификации ES2015, где сказано что у стрелочных функций нет своего контекста, в них **this** ссылается на тот же объект, который является контекстом для функции выше, в которой эта функция объявлена. Получается в нашем случае **this** внутри методов будет ссылаться на глобальный объект, и методы **call**, **apply** и **bind** не помогут нам переопределить этот контекст. Казалось бы, как не удобно, но на самом деле стрелочные функции очень удобны когда используются как анонимные функции внутри методов, т.к. мы можем использовать в них **this** из самого метода. Давайте посмотрим на примере:

Листинг 18.

```
const Stand = {
   model: "Stand-1",
   robots: ['Roomba-1', 'Tango-1', 'Samba-1', 'Roomba-2'],
    // Метод, с использованием стрелочной функции в качестве
функции обратного вызова.
   startTestingArrow: function() {
     console.log('Start testing...');
     this.robots.forEach((value) => {
       console.log('stand: ', this.model, 'is testing robot: ',
value);
     })
   },
   // Метод, с использованием классической функции в качестве
функции обратного вызова.
   startTestingClassic: function() {
     console.log('Start testing...');
     this.robots.forEach(function(value) {
       console.log('stand: ', this.model, 'is testing robot: ',
value);
     })
   },
 };
  Stand.startTestingArrow();
 // Start testing...
 // stand:  Stand-1 is testing robot:  Roomba-1
 // stand:  Stand-1 is testing robot:  Tango-1
 // stand:  Stand-1 is testing robot:  Samba-1
 // stand:  Stand-1 is testing robot:  Roomba-2
  Stand.startTestingClassic();
 // Start testing...
 // stand:  undefined is testing robot:  Roomba-1
 // stand:  undefined is testing robot:  Tango-1
 // stand:  undefined is testing robot:  Samba-1
 // stand:  undefined is testing robot:  Roomba-2
```

Как мы можем видеть, первый метод, в котором функция обратного вызова объявлена через стрелочную функцию отработал отлично, т.к. не имея своего собственного **this** функция обратного вызова использовала **this** из метода объекта, потому получила доступ к свойству **model**. А вот второй метод, в котором функция обратного вызова была объявлена классическим способом не смогла получить доступ к методу объекта, т.к. у неё есть свой собственный **this**, который был не определен в момент вызова этой функции.

Объект через class

Давайте рассмотрим вариант создания объекта через ключевое слово **class** и как осуществляется привязка контекста к методам в таком случае. Начнем с примера создания нашего робота пылесоса, пока без привязок контекста:

Листинг 19.

```
// Класс робот-пылесос.
class RobotVacuumCleaner {
    // Свойства класса.
    model = "Romba-1";
    power = 200;
    batterySize = 2100;
    boxSize = 0.5;
    workTime = 45;
    counterOfStarts = 0;
    isFull = false;
    isObstacle = false;
    isUVLampOn = false;

    // Конструктор класса, мы изучим его подробнее на следующем
уроке.
    constructor() {
    }

    // Методы класса.
    startCleaning() {
        this.counterOfStarts++;
        console.log('I am cleaning... I have been started: ',
this.counterOfStarts, 'times.');
    }

    goCharge() {
        console.log('I am going to charge...');
    }

    switchUVLamp() {
        this.isUVLampOn = !this.isUVLampOn;
        console.log(`UV lamp is ${this.isUVLampOn ? 'working' :
'not working'}.`);
    }
}
```

```
// Создадим экземпляр класса.
const Roomba = new RobotVacuumCleaner();
```

Все очень схоже с обычным объектом - те же свойства, и методы, только добавился конструктор класса (мы рассмотрим его на следующем уроке) и для создания самого робота нам необходимо вызвать наш класс с использованием ключевого слова new. Давайте попробуем обратиться к свойствам и методам робота с использованием `setTimeout`:

Листинг 20.

```
// Обращение к свойствам объекта.
console.log(Roomba.model); // "Romba-1"
console.log(Roomba.isFull); // false

// Отложенный вызов методов объекта.
setTimeout(Roomba.startCleaning, 1000);

// Установим свойства объекта isUVLampOn в true, чтобы
продемонстрировать результат работы метода switchUVLamp.
Roomba.isUVLampOn = true;

setTimeout(Roomba.switchUVLamp, 2000);

setTimeout(Roomba.goCharge, 3000);

// I am cleaning... I have started:  NaN times.
// UV lamp is working.
// I am going to charge...
```

Получили все то же самое, что мы получали и при работе с простым объектом. Вызов методов класса, которые были переданы как функции обратного вызова в метод `setTimeout`, теряют свой контекст, и **`this`** в них начинает ссылаться на глобальный объект. Но для решения этой проблемы, при использовании классов мы можем просто привязать контекст к методам еще на этапе создания класса, в конструкторе, используя метод **`bind`**.

Листинг 21.

```
// Класс робот-пылесос.
class RobotVacuumCleaner {
    // Свойства класса.
    model = "Romba-1";
    power = 200;
    batterySize = 2100;
    boxSize = 0.5;
    workTime = 45;
    counterOfStarts = 0;
    isFull = false;
    isObstacle = false;
    isUVLampOn = false;

    // Используем его, чтобы привязать все методы класса к
контексту - текущему объекту (this).
    constructor() {
        this.startCleaning = this.startCleaning.bind(this);
        this.goCharge = this.goCharge.bind(this);
        this.switchUVLamp = this.switchUVLamp.bind(this);
    }

    // Методы класса.
    startCleaning() {
        this.counterOfStarts++;
        console.log('I am cleaning... I have been started: ',
this.counterOfStarts, 'times.');
    }

    goCharge() {
        console.log('I am going to charge...');
    }

    switchUVLamp() {
        this.isUVLampOn = !this.isUVLampOn;
        console.log(`UV lamp is ${this.isUVLampOn ? 'working' :
'not working'}.`);
    }
}

// Создадим экземпляр класса.
const Roomba = new RobotVacuumCleaner();
```

Просто, не правда ли? Достаточно перезаписать ссылки на методы класса, на их привязанную к контексту версию. Конструкция выглядит так:

```
this.<метод класса> = this.<метод класса>.bind(this).
```

Теперь мы можем снова вызвать наши методы с использованием `setTimeout`:

Листинг 22.

```
// Обращение к свойствам объекта.
console.log(Roomba.model); // "Romba-1"
console.log(Roomba.isFull); // false

// Отложенный вызов методов объекта.
setTimeout(Roomba.startCleaning, 1000);

// Установим свойства объекта isUVLampOn в true, чтобы
продемонстрировать результат работы метода switchUVLamp.
Roomba.isUVLampOn = true;

setTimeout(Roomba.switchUVLamp, 2000);

setTimeout(Roomba.goCharge, 3000);

// I am cleaning... I have started: 1 times.
// UV lamp is not working.
// I am going to charge...
```

И есть способ еще проще: объявить наши методы через стрелочную функцию, как мы помним, стрелочная функция не имеет своего контекста, поэтому будет использовать контекст функции, внутри которой объявлена стрелочная функция, а так как мы используем класс для создания объекта, то он будет являться контекстом внутри стрелочной функции:

Листинг 23.

```
// Класс робот-пылесос.
class RobotVacuumCleaner {
    // Свойства класса.
    model = "Romba-1";
    power = 200;
    batterySize = 2100;
    boxSize = 0.5;
    workTime = 45;
    counterOfStarts = 0;
    isFull = false;
    isObstacle = false;
    isUVLampOn = false;

    // Конструктор класса, мы изучим его подробнее на следующем
уроке.
    constructor() {
    }

    // Методы класса.
    startCleaning = () => {
        this.counterOfStarts++;
        console.log('I am cleaning... I have been started: ',
this.counterOfStarts, 'times.');
    }

    goCharge = () => {
        console.log('I am going to charge...');
    }

    switchUVLamp = () => {
        this.isUVLampOn = !this.isUVLampOn;
        console.log(`UV lamp is ${this.isUVLampOn ? 'working' :
'not working'}.`);
    }
}

// Создадим экземпляр класса.
const Roomba = new RobotVacuumCleaner();
```

Попробуем вызвать наши методы:

Листинг 24.

```
// Обращение к свойствам объекта.
console.log(Roomba.model); // "Romba-1"
console.log(Roomba.isFull); // false

// Отложенный вызов методов объекта.
setTimeout(Roomba.startCleaning, 1000);

// Установим свойства объекта isUVLampOn в true, чтобы
продемонстрировать результат работы метода switchUVLamp.
Roomba.isUVLampOn = true;

setTimeout(Roomba.switchUVLamp, 2000);

setTimeout(Roomba.goCharge, 3000);

// I am cleaning... I have started: 1 times.
// UV lamp is not working.
// I am going to charge...
```

Подведем итоги

Любая функция имеет указатель **this** на свой контекст в момент вызова. Внутри объекта это обычно сам объект, но как мы могли видеть, если метод вызывается в отрыве от объекта, то этот указатель начинает ссылаться на глобальный объект, или принимает значение undefined, при использовании строгого режима в коде (**“use strict”**). Для решения таких ситуаций у каждой функции есть три метода **call**, **apply** и **bind**, которые позволяют привязать нужный нам контекст к функции во время её вызова (**call**, **apply**) или навсегда (**bind**).

Домашнее задание

Задание 1. "Управление библиотекой книг"

Реализуйте класс `Book`, представляющий книгу, со следующими свойствами и методами:

- Свойство title (название) - строка, название книги.
- Свойство author (автор) - строка, имя автора книги.
- Свойство pages (количество страниц) - число, количество страниц в книге.

- Метод displayInfo() - выводит информацию о книге (название, автор и количество страниц).

Задание 2. "Управление списком студентов"

Реализуйте класс Student, представляющий студента, со следующими свойствами и методами:

- Свойство name (имя) - строка, имя студента.
- Свойство age (возраст) - число, возраст студента.
- Свойство grade (класс) - строка, класс, в котором учится студент.
- Метод displayInfo() - выводит информацию о студенте (имя, возраст и класс).
- javascript

```
// Пример использования класса
const student1 = new Student("John Smith", 16, "10th grade");
student1.displayInfo();
// Вывод:
// Name: John Smith
// Age: 16
// Grade: 10th grade

const student2 = new Student("Jane Doe", 17, "11th grade");
student2.displayInfo();
// Вывод:
// Name: Jane Doe
// Age: 17
// Grade: 11th grade
```