

# Объектно-ориентированное программирование и наследование

## Курс JavaScript про ECMAScript



# Оглавление

|  |           |
|--|-----------|
| <b>Введение</b>  | <b>2</b>  |
| <b>Прототип</b>  | <b>3</b>  |
| Методы для установки прототипа.  | 8         |
| getPrototypeOf   | 8         |
| setPrototypeOf   | 8         |
| Конструктор объекта  | 11        |
| Оператор new   | 13        |
| <b>Object.create</b>   | <b>15</b> |
| <b>Создание объектов и наследование с использованием class и extends</b> | <b>16</b> |

## Введение

На прошлом уроке мы узнали, что при написании алгоритмов очень удобно использовать объектный подход, когда для сущностей реального мира в языке программирования создаются их аналоги в виде объектов со свойствами и методами. Так мы создавали объекты роботов пылесосов, методы которых позволяли им функционировать. На этом уроке мы продолжим использовать эти же примеры и разберем подробнее тему объектно-ориентированного программирования.

Ранее мы создавали три модели роботов: Roomba, Tango и Samba. По сути они отличались только свойствами, и мы дублировали много кода для их создания. Вот тут нам как раз могло помочь наследование. Мы можем выделить все базовые функции и свойства (которые одинаковы для всех моделей) в некоторый базовый объект `RobotVacuumCleaner` - он будет являться родителем для всех более специфичных моделей, а они по отношению к нему будут называться потомками. Потомки наследуют все свойства и методы от родителей и добавляют к ним свои собственные, а некоторые могут переписать под свои нужды. Мы рассмотрим это

все на примерах чуть позже, сначала нам нужно познакомиться с тем как работает наследование в JavaScript и что такое прототип объекта.

## Прототип

До ES2015 в JavaScript не было понятия **class** как в других языках программирования, поэтому возможности для работы с объектами как в объектно-ориентированных языках были реализованы через использование прототипов - специализированного свойства в объекте, которое добавляется к любому новому объекту. Так наследование свойств от родителя к потомку происходит через это специальное свойство `__proto__`, которое указывает на какой объект ссылаться, как на родитель.

Давайте на примерах рассмотрим что такое прототип, зачем это нужно и как работает. Для примеров возьмем примеры из нашего прошлого урока про роботов-пылесосов.

Для начала создадим просто объект робота пылесоса без характеристик и с обобщенными методами. Это будет некий абстрактный робот-пылесос в вакууме:

Листинг 1.

```
// Объект робот-пылесос.
const VacuumCleaner = {
  Model: "vacuum cleaner",
  counterOfStarts: 0,
  isFull: false,
  isObstacle: false,

  startCleaning: function () {
    this.counterOfStarts++;
    // Добавим дополнительный вывод, чтобы знать чей метод
    мы вызвали.
    console.log('I am the method of VacuumCleaner');
    console.log('I am cleaning... I have been started: ',
this.counterOfStarts, 'times.');
```

```
  },

  goCharge: function () {
    // Добавим дополнительный вывод, чтобы знать чей метод
    мы вызвали.
    console.log('I am the method of VacuumCleaner');
    console.log('I am going to charge...');
```

```
    }  
};
```

Мы оставили в нем только служебные свойства и методы, при этом мы убрали даже свойства `isUVLampOn` - так как это свойство будет не во всех моделях пылесосов. Теперь мы хотели бы создать пылесос с конкретными характеристиками, но чтобы не создавать объект с нуля и прописывать все свойства мы можем взять базовую модель `VacuumCleaner` и наследоваться от неё, установив у нового объекта свойства прототипа `__proto__` на родительский объект и добавив новые свойства - вот так:

#### Листинг 2.

```
// Объявление родительского объекта смотри в листинге 1.  
// Объект робот-пылесос.  
const DancingSeries = {  
  // Объявляем новые свойства и переопределять свойство model.  
  model: "dancing series",  
  power: 200,  
  batterySize: 2100,  
  boxSize: 0.5,  
  workTime: 45,  
  isUVLampOn: false,  
  // Добавляем новый метод.  
  switchUVLamp: function () {  
    // Добавим дополнительный вывод, чтобы знать чей метод  
мы вызвали.  
    console.log('I am the method of DancingSeries');  
    this.isUVLampOn = !this.isUVLampOn;  
    console.log(`UV lamp is ${this.isUVLampOn ? 'working' :  
'not working'}`);  
  },  
  
  // Делаем ссылку на прототип от родителя.  
  __proto__: VacuumCleaner,  
};
```

Наш новый объект **DancingSeries** - это тоже некий общий объект для создания серии пылесосов с одинаковым функционалом, и разными характеристиками.

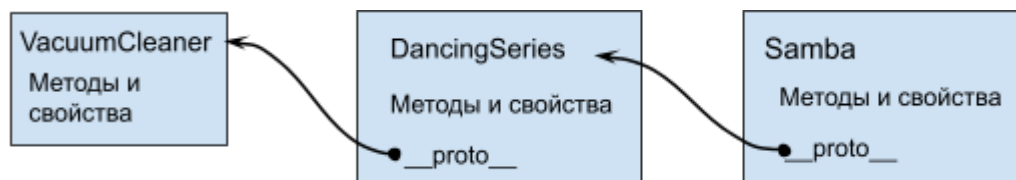
Далее на основе него мы уже можем создать несколько конкретных моделей пылесосов, переписав в них характеристики. Давайте создадим робот **Samba**:

Листинг 3.

```
// Объявление базового родительского объекта смотри в листинге 1.
// Объявление DancingSeries смотри в листинге 2.
// Объект робот-пылесос.
const Samba = {
  // Обновляем свойства под конкретную модель.
  model: "Samba-1",
  power: 250,
  batterySize: 2500,
  workTime: 50,

  // Делаем ссылку на прототип от родителя.
  __proto__: DancingSeries,
};
```

Как мы можем видеть, код конкретного объекта стал очень маленьким, и мы получили цепочку объектов, связанных через прототипы благодаря свойству `__proto__`.



Наш первый родительский объект тоже имеет свойство `__proto__`, оно ссылается на объект `Object`, т.к. мы не указывали его. Давайте попробуем вызвать методы и свойства нашего нового объекта `Samba`:

Листинг 4.

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

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

Samba.isUVLampOn = true;
Samba.switchUVLamp();
// I am the method of DancingSeries
// 'UV lamp is not working.'

Samba.goCharge();
// I am the method of VacuumCleaner
// 'I am going to charge...'
```

Все свойства и методы доступны при обращении к объекту Samba, и как мы можем видеть, когда их нет в самом объекте движок JavaScript пытается найти их в родительском объекте, который указан в свойстве `__proto__`, и будет искать по цепочки прототипов пока не найдёт, либо получит `undefined`, если свойство или метод не будет найдено даже в самом верхнем объекте `Object`.

Именно механизм прототипов позволяет нам использовать методы объекта `Array`, когда мы создаем свои пользовательские массивы, для них свойство `__proto__` автоматически выставляется на объект `Array`, а `__proto__` объекта `Array` ссылается на объект `Object`.

Если мы попытаемся в потомке переопределить свойство или метод из родительского объекта, то родительский объект остается нетронутым, а в дочерний запишется новое свойство или метод, и оно будет вызываться при обращении к нему, давайте создадим еще одного робота, и переопределим в нем метод **`startCleaning`**:

Листинг 5.

```
// Объект робот-пылесос.
const Djaiv = {
  // Обновляем свойства под конкретную модель.
  model: "Djaiv-1",
  power: 250,
  batterySize: 2500,
  workTime: 50,

  // Переопределим метод startCleaning.
  startCleaning: function () {
    this.counterOfStarts++;
    // Добавим дополнительный вывод, чтобы знать чей метод
    // мы вызвали.
    console.log('I am the method of Djaiv');
    console.log('I am Djaiv, and I am cleaning... I have
    been started: ', this.counterOfStarts, 'times.');
```

И попробуем вызвать методы пылесоса Samba и Djaiv, чтобы увидеть разницу:

#### Листинг 6.

```
// Объявление базового родительского объекта смотри в листинге
1.
// Объявление DancingSeries смотри в листинге 2.
// Объект Samba смотри в листинге 3.
// Объект Djaiv смотри в листинге 5.

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

Djaiv.startCleaning();
// I am the method of Djaiv
// I am Djaiv, and I am cleaning... I have been started: 1
times.
```

Мы можем видеть, что объект Samba использует родительский метод и он не изменился, а Djaiv использует свой собственный метод.

# Методы для установки прототипа.

Устанавливать прототип объекта можно используя свойство `__proto__`, но также в языке есть два метода для чтения и установки прототипа объекта - это `getPrototypeOf` и `setPrototypeOf`. Эти методы не доступны в браузере Internet Explorer версии ниже 10.

## getPrototypeOf

Метод `getPrototypeOf` позволяет получить ссылку на объект прототип. Давайте узнаем какой объект является прототипом для нашего объекта `Djaiv`, потом посмотрим кто является его прототипом и кто является прототипом его прототипа:

Листинг 9.

```
// Получим прототип для объекта Djaiv.
const DjaivProto = Object.getPrototypeOf(Djaiv);
console.log(DjaivProto.model); // dancing series

const DjaivProtoProto = Object.getPrototypeOf(DjaivProto);
console.log(DjaivProtoProto.model); //vacuum cleaner

const DjaivProtoProtoProto =
Object.getPrototypeOf(DjaivProtoProto);
console.log(DjaivProtoProtoProto); // [object Object]
```

В последнем прототипе мы не стали смотреть свойство `model`, т.к. там его нет, мы добрались до самого высокого родителя, которым является объект `Object`, все объекты наследуются от него. Если мы попытаемся получить его прототип, то в ответ получим `null`, т.к. Объект `Object` не имеет прототипа.

## setPrototypeOf

Зачем может понадобиться знать прототип объекта? Давайте создадим другую серию роботов - `musicSeries`, с немного другим функционалом (они будут уметь мыть полы), и установим её в качестве прототипа для нашего нового пылесоса `Blues`. Так как новая серия будет иметь дополнительный функционал, которого нет в серии `DancingSeries`, прежде чем вызвать такой функционал у определенного объекта, стоит проверить, кто его родитель. Для установки прототипа используем метод `setPrototypeOf` - он принимает два аргумента, первый это объект для



которого устанавливается прототип, второй - это объект который будет прототипом для первого.

Для начала создадим объект серии:

#### Листинг 10.

```
// Объект робот-пылесос.
const MusicSeries = {
  // Объявляем новые свойства и переопределяем свойство model.
  model: "music series",
  power: 200,
  batterySize: 2100,
  boxSize: 0.5,
  workTime: 45,

  // Добавляем новый метод.
  startWipe: function () {
    // Добавим дополнительный вывод, чтобы знать чей метод
    мы вызвали.
    console.log('I am the method of MusicSeries');
    console.log('I am starting to wipe the floor...');
  },

  // Делаем ссылку на прототип от родителя.
  __proto__: VacuumCleaner,
};

// Объект робот-пылесос.
const MusicSeries = {
  // Объявляем новые свойства и переопределяем свойство model.
  model: "music series",
  power: 200,
  batterySize: 2100,
  boxSize: 0.5,
  workTime: 45,

  // Добавляем новый метод.
  startWipe: function () {
    // Добавим дополнительный вывод, чтобы знать чей метод
    мы вызвали.
    console.log('I am the method of MusicSeries');
    console.log('I am starting to wipe the floor...');
  },

  // Делаем ссылку на прототип от родителя.
  __proto__: VacuumCleaner,
```

```
};
```

Создадим нашего нового робота:

Листинг 11.

```
// Объект робот-пылесос.
const Blues = {
  // Обновляем свойства под конкретную модель.
  model: "Bluees-1",
  power: 250,
  batterySize: 2500,
  workTime: 50,
};

// Установим прототип для робота.
Object.setPrototypeOf(Blues, MusicSeries);
```

Теперь можем попробовать вызвать методы наших роботов, проверяя кто является их прототипом:

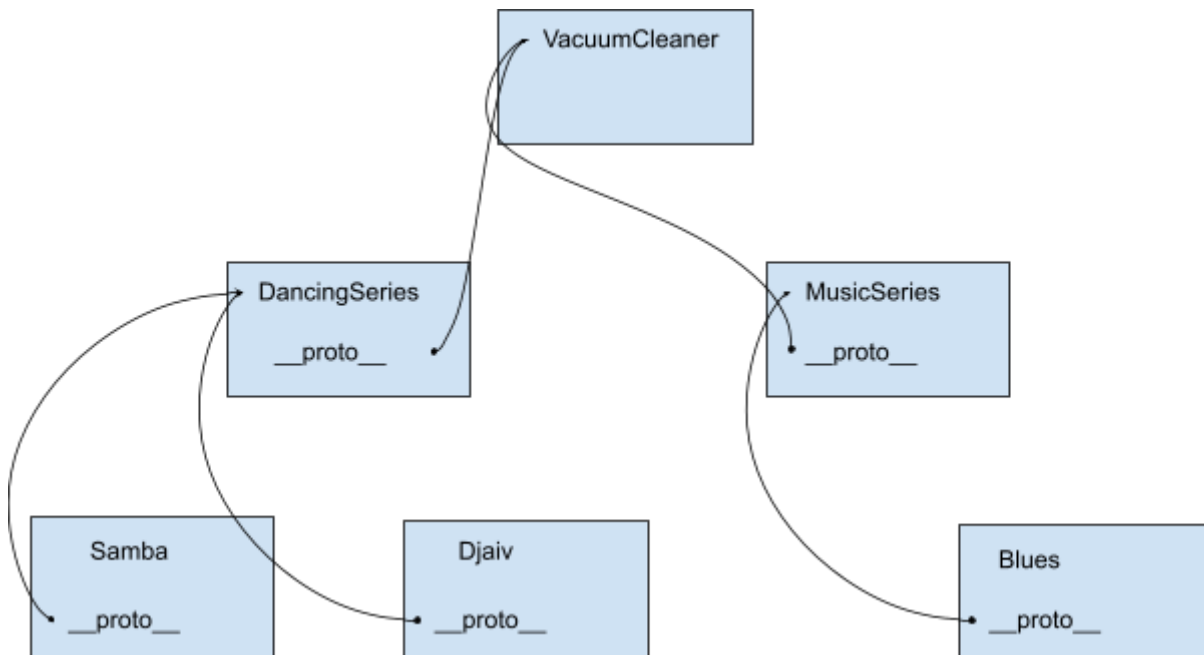
Листинг 12.

```
// Объявление базового родительского объекта смотри в листинге 1.
// Объявление DancingSeries смотри в листинге 2.
// Объект Djaiv смотри в листинге 5.
// Объявление MusicSeries смотри в листинге 10.
// Объект Blues смотри в листинге 11.
if (Object.getPrototypeOf(Djaiv).model === 'dancing series') {
  Djaiv.startCleaning(); //
}

if (Object.getPrototypeOf(Blues).model === 'music series') {
  Blues.startWipe(); //
}

// Если мы не будем проверять прототип и просто вызовем метод
чужого прототипа мы, естественно, получим ошибку.
Djaiv.startWipe(); // Uncaught TypeError: Djaiv.startWipe is
not a function
```

Давайте посмотрим на структуру созданных нами базовых моделей и всех роботов что мы уже создали, чтобы лучше увидеть наследование:



## Конструктор объекта

До этого момента мы создавали объекты и устанавливали прототип для одного конкретного объекта. А что если мы хотим создать сразу пять объектов одного типа, например пять пылесосов Samba, неужели нам придется писать однотипный код каждого объекта? К счастью, как и во многих других языках программирования, в языке JavaScript предусмотрели возможность создавать конструктор объекта - функцию, которая позволяет создавать экземпляры объектов, при этом позволяя произвести инициализацию объекта. Конструктор объекта вызывается не напрямую, а с помощью оператора **new**, который позволяет создать новый экземпляр объекта.

Чтобы создать функцию конструктор для объекта, она должна называться с большой буквы (не обязательно, но это позволяет явно видеть что это конструктор для объекта), а внутри этой функции через **this** объявить свойства и методы для объекта. При этом мы можем передавать аргументы этой функции, которые можем использовать как первоначальные значения для свойств, или для создания сложной логики в методах, или даже определять какие методы и свойства получит объект в зависимости от аргументов.

Давайте создадим функцию конструктор для роботов Samba:

### Листинг 13.

```
// Объявление DancingSeries смотри в листинге 2.

// Конструктор объекта робот-пылесос.
function Samba(serialNumber) {
    // Создаем свойства объекта, используя this.
    this.serialNumber = serialNumber;
    this.model = "Samba-1";
    this.power = 250;
    this.batterySize = 2500;
    this.workTime = 50;

    // Делаем ссылку на прототип от родителя.
    this.__proto__ = DancingSeries;
}

// Создадим экземпляр нового объекта.
const Samba1 = new Samba(1014778);

console.log(Samba1.serialNumber); // 1014778
console.log(Samba1.startCleaning()); // I am the method of
VacuumCleaner
// I am cleaning... I have been started: 1 times.
```

Вместо создания конкретного объекта, мы создали функцию, которая делает все то же самое, только записывает свойства и методы через **this**. А чтобы создать непосредственно сам объект, нам нужно вызвать эту функцию через оператор **new**. Почему именно через него и что он делает мы разберем ниже, а пока давайте создадим несколько экземпляров робота Samba, чтобы увидеть всю прелесть конструкторов:

### Листинг 14.

```

// Объявление базового родительского объекта смотри в листинге
1.
// Объявление DancingSeries смотри в листинге 2.
// Конструктор объекта робот-пылесос Samba смотри в листинге
13.

// Создадим 10 роботов пылесосов Samba, как на конвейере.
const robots = [];

for (let index = 0; index < 10; index++) {
    // Создадим экземпляр нового объекта и добавляем его в
    массив наших роботов, каждый с уникальным серийным номером.
    robots.push(new Samba(index));
}

console.log(robots[3].serialNumber); // 3
console.log(robots[7].serialNumber); // 7

```

Далее мы можем взаимодействовать с нашим массивом роботов как нам удобно, обрабатывать его в циклах, обращаться к каждому роботу по отдельности, и нам понадобилось всего четыре строчки кода чтобы создать такое количество роботов.

Затронем еще момент об установке прототипа для объектов создаваемых конструктором. Мы можем указывать `this.__proto__` для установки прототипа, а можем указать наш прототип в `prototype` свойстве самого конструктора вот так:

#### Листинг 15.

```

// Объявление базового родительского объекта смотри в листинге
1.
// Объявление DancingSeries смотри в листинге 2.

// Конструктор объекта робот-пылесос.
function Samba(serailNumber) {
    // Создаем свойства объекта, используя this.
    this.serialNumber = serailNumber;
    this.model = "Samba-1";
    this.power = 250;
    this.batterySize = 2500;
    this.workTime = 50;
}

// Делаем ссылку на прототип от родителя.
Samba.prototype = DancingSeries;

```

```
// Создадим экземпляр нового объекта.  
const Samba1 = new Samba(1014778);  
  
console.log(Samba1.serialNumber); // 1014778  
console.log(Samba1.startCleaning()); // I am the method of  
VacuumCleaner  
// I am cleaning... I have been started: 1 times.
```

Давайте теперь разберемся почему наша функция, которая ничего не возвращает в коде после вызова через оператор `new` возвращает нам новый объект со всеми нужными нам свойствами. А делает это не сама функция, а оператор **`new`**.

## Оператор `new`

Оператор `new` позволяет создавать новые объекты, используя для этого функцию-конструктор. Работает он следующим образом:

1. Создает пустой объект, который наполнит всем необходимым.
2. Устанавливает этот объект как **`this`** для функции конструктора, чтобы можно было использовать **`this`** внутри функции и добавлять свойства и методы в этот объект.
3. Вызывает функцию конструктор для инициализации объекта.
4. Если у функции конструктора есть свойство **`prototype`**, устанавливает значение этого свойства как прототип для нового объекта (свойство **`__proto__`**).
5. Устанавливает свойство **`constructor`** объекта ссылкой на функцию конструктор.
6. Если функция конструктор не возвращает ничего или возвращает какое-то примитивное значение, то оператор `new` вернет новый созданный и наполненный объект, если конструктор возвращает объект, то оператор **`new`** вернет этот объект.

Не так сложно, давайте попробуем создать свою версию оператора `new` в виде функции:

Листинг 16.

```

// Наша реализация оператора new через функцию createObject.
function createObject(constructor) {
    // Создаем новый объект.
    const obj = {};

    // Установим новому объекту прототипом прототип
    функции-конструктора
    Object.setPrototypeOf(obj, constructor.prototype);

    // Вызовем функцию-конструктор, передав ей как this
    созданный на шаге 1 объект, и передадим остальные аргументы,
    если они были переданы в createObject
    const argsArray = Array.prototype.slice.apply(arguments);
    const result = constructor.apply(obj, argsArray.slice(1));

    // Вернем новый объект, если конструктор вернул примитивное
    значение или undefined, иначе вернем то, что вернул
    конструктор.

    if (!result || typeof result === 'string' || typeof result
    === 'number' || typeof result === 'boolean') {
        return obj
    } else {
        return result;
    }
}

// Создадим экземпляр нового объекта.
const Sambal = createObject(Samba, 1014778);
// Проверим установку свойств в конструкторе.
console.log(Sambal.serialNumber); // 1014778
// Проверим, что прототип установился корректно, и мы можем
вызывать методы из родительских объектов.
console.log(Sambal.__proto__); // {model: "dancing series",
power: 200, batterySize: 2100, boxSize: 0.5, workTime: 45, ...}
console.log(Sambal.startCleaning()); // I am the method of
VacuumCleaner
// I am cleaning... I have been started: 1 times.

// Проверим присвоение конструктора.
console.log(Sambal.constructor); // function Object() { [native
code] }

```

! Свойство `__proto__` объекта и свойство `prototype` у функции конструктора это не одно и то же. Свойство `__proto__` есть у экземпляра объекта, и оно позволяет находить родителей объекта, свойство `prototype` выполняет служебную функцию при создании экземпляра объекта через оператор `new`.

Посмотрите на пример кода, который расставит точки на «i».

```
// Конструктор объекта робот-пылесос.
function Samba(serialNumber) {
  // Создаем свойства объекта, используя this.
  this.serialNumber = serialNumber;
  this.model = "Samba-1";
  this.power = 250;
  this.batterySize = 2500;
  this.workTime = 50;
}

// Делаем ссылку на прототип от родителя.
Samba.prototype = DancingSeries;

// Создадим экземпляр нового объекта.
const Sambal = new Samba(1014778);
// Посмотрим на свойства __proto__ и prototype
console.log(Sambal.__proto__); // {model: "dancing series",
power: 200, batterySize: 2100, boxSize: 0.5, workTime: 45, ...}
console.log(Sambal.prototype); // undefined
console.log(Sambal.__proto__ === Samba.prototype); // true
```

! В экземпляре объекта нет свойства `prototype`, данные из него перешли в свойство `__proto__`.

Мы также можем создавать новые объекты с помощью метода `create`, доступного в объекте `Object`.

## Object.create

Метод `Object.create` позволяет создавать новые объекты, принимая в качестве аргументов объект прототип для создаваемого объекта, и вторым аргументом (необязательным) свойства для нового объекта в формате объект с ключами и значениями дескрипторов для свойств. Использовать этот метод для создания



новых объектов по типу наших роботов-пылесосов не очень удобно, нужно будет либо все свойства нового объекта указывать дескрипторами, что не очень удобно (но гибко), либо создавать с помощью метода объект с указанием прототипа, а все остальные свойства добавлять позже, но есть важная особенность у этого метода, которую можно использовать - в качестве первого аргумента можно передать `null` и тогда будет создан объект без прототипа. Давайте посмотрим на примере, зачем нам это нужно:

Листинг 17.

```
// Объявление базового родительского объекта смотри в листинге 1.
// Объявление DancingSeries смотри в листинге 2.
// Конструктор объекта робот-пылесос Samba смотри в листинге 13.

// Создадим робот пылесосSamba.
const Samba1 = new Samba(101);

// Попробуем обратиться к стандартному методу toString, хоть мы его и не объявляли ни в одном из объектов.
console.log(Samba1.toString()); // [object Object]
```

Хоть мы и не объявляли метод `toString` в нашей цепочки объектов, но он присутствует и идёт от самого первого объекта (базового), т.к. его прототип - это сам объект `Object`, и метод `toString` пришел от него. Иногда бывает так, что нам совсем не нужны чужие методы, и не нужен прототип в объекте, потому что обращение к свойствам объекта определяются тем, что напишет пользователь, и пользователь может запросить свойство `toString`, которое мы не хотели бы показывать, и вот тут как раз можно использовать метод `Object.create`:

Листинг 18.

```
// Создадим пустой объект без прототипа.
const Samba1 = Object.create(null);

// Попробуем обратиться к стандартному методу toString и посмотреть на свойство __proto__
console.log(Samba1.toString); // undefined
console.log(Samba1.__proto__); // undefined
```

Таким образом мы получили чистый объект, без прототипа, а следовательно и без свойств из него.

# Создание объектов и наследование с использованием class и extends

Мы научились создавать объекты и устанавливать прототипы, создавать конструкторы и разобрались как работает оператор new. С приходом ES2015 в язык был добавлен синтаксис классов, чтобы все эти операции можно было делать удобнее и в более привычном синтаксисе для тех, кто уже программировал с использованием классов в других языках программирования. Давайте создадим наших роботов с использованием нового синтаксиса.

Листинг 19.

```
// Класс робот-пылесос.
class VacuumCleaner {
  model = "vacuum cleaner";
  counterOfStarts = 0;
  isFull = false;
  isObstacle = false;
  // Для создания конструктора, нужно создать метод
  constructor.
  constructor() {
  }

  startCleaning() {
    this.counterOfStarts++;
    // Добавим дополнительный вывод, чтобы знать чей метод
    мы вызвали.
    console.log('I am the method of VacuumCleaner');
    console.log('I am cleaning... I have been started: ',
this.counterOfStarts, 'times.');
```

```
  }

  goCharge() {
    // Добавим дополнительный вывод, чтобы знать чей метод
    мы вызвали.
    console.log('I am the method of VacuumCleaner');
    console.log('I am going to charge...');
```

```
  }
}

// Попробуем создать экземпляр класса и посмотреть как он
работает.
const BaseRobot = new VacuumCleaner;
```

```

console.log(BaseRobot.constructor); // class VacuumCleaner {
//  model = "vacuum cleaner";
//  counterOfStarts = 0;
//  isFull = false;
//  isObstacle = false;
//  Для создания конструктора, нужно создать метод constructor.
//  constructor() {
//  }
//  ...

console.log(BaseRobot.model); // vacuum cleaner
console.log(BaseRobot.startCleaning()); // I am the method of
VacuumCleaner
// I am cleaning... I have been started: 1 times.

```

Мы создали наш базовый класс, и можем видеть, что у него есть методы и свойства, а метод конструктор распечатывает сам класс. Также класс устанавливает прототип для объекта, для этого нужно расширить наш базовый класс с помощью ключевого слова `extend`. Давайте создадим наш расширяющий класс `DancingSeries`:

#### Листинг 20.

```

// Объявление родительского класса смотри в листинге 19.
// Расширенный класс DancingSeries. С помощью extends мы
указываем от какого класса будем наследоваться.
class DancingSeries extends VacuumCleaner {
    // Объявляем новые свойства и переопределяем свойство model.
    model = "dancing series";
    power = 200;
    batterySize = 2100;
    boxSize = 0.5;
    workTime = 45,
    isUVLampOn = false;

    // Добавляем новый метод.
    switchUVLamp() {
        // Добавим дополнительный вывод, чтобы знать чей метод
        мы вызвали.
        console.log('I am the method of DancingSeries');
        this.isUVLampOn = !this.isUVLampOn;
        console.log(`UV lamp is ${this.isUVLampOn ? 'working' :
        'not working'}.`);
    }
};

```

```
// Создадим новый экземпляр класса, чтобы посмотреть как он
// работает и что в нем есть.
const DancingRobot = new DancingSeries;
console.log(DancingRobot.__proto__); // VacuumCleaner
{constructor: f, switchUVLamp: f}
console.log(DancingRobot.model); // dancing series
console.log(DancingRobot.switchUVLamp()); // I am the method of
DancingSeries
// lamp is working.
```

Мы расширили наш базовый класс, и создав экземпляр нового робота видим в нём прототип на базовый класс, новый метод и предопределенное свойство model. Все работает как должно и ожидается.

Давайте теперь разберем что же делают классы. Работа классов достаточно проста и прозрачна. Объявляя класс, движок JavaScript создает функцию конструктор по имени класса и берет её код из метода constructor класса, если такого метода нет, то функция будет пустой. Если класс расширяет другой класс, то для этой функции указывается свойство prototype. После чего находит все остальные методы объекта и прописывает их в свойство prototype для новой функции. Вот почему мы в прототипе для объекта DancingSeries увидели не только методы родительского класса, но и метод класса DancingSeries. Ну а дальше все просто, мы получили функцию конструктор, с прототипом и всеми необходимыми методами, поэтому при вызове оператора new мы создаем наш объект, как будто он был создан старым способом. Конечно в работе классов есть тонкости, например функцию конструктор, созданную классом, нельзя вызвать без использования оператора new, а все создаваемые ей методы будут помечены как неперечислимые.

Давайте возьмем предыдущий пример, и посмотрим на функцию конструктор, созданную классом:

#### Листинг 21.

```
// Объявление родительского класса смотри в листинге 19.
// Расширенный класс DancingSeries смотри в листинге 20.
// Созданный конструктор является функцией, код которой взят из
// конструктора.
console.log(DancingSeries ===
DancingSeries.prototype.constructor); // true
// В созданном конструкторе есть свойство prototype и оно
// содержит все методы.
console.log(DancingSeries.prototype); // VacuumCleaner
{constructor: f, switchUVLamp: f}
```

Синтаксис классов намного проще и лучше читается, а для многих программистов, кто пришел к изучению JavaScript после других объектно-ориентированных языков он намного привычнее. К сожалению еще не все браузеры умеют работать с синтаксисом классов напрямую, поэтому чаще всего в проектах необходимо использовать дополнительные инструменты для конвертации кода ES2015 в более привычный для браузеров ES5.

## Домашнее задание

### Задание 1. "Управление персоналом компании"

Реализуйте класс `Employee` (сотрудник), который имеет следующие свойства и методы:

- Свойство `name` (имя) - строка, имя сотрудника.
- Метод `displayInfo()` - выводит информацию о сотруднике (имя).

Реализуйте класс `Manager` (менеджер), который наследует класс `Employee` и имеет дополнительное свойство и метод:

- Свойство `department` (отдел) - строка, отдел, в котором работает менеджер.
- Метод `displayInfo()` - переопределяет метод `displayInfo()` родительского класса и выводит информацию о менеджере (имя и отдел).

```
// Пример использования классов
const employee = new Employee("John Smith");
employee.displayInfo();
// Вывод:
// Name: John Smith

const manager = new Manager("Jane Doe", "Sales");
manager.displayInfo();
// Вывод:
// Name: Jane Doe
// Department: Sales
```

### Задание 2. "Управление списком заказов"

Реализуйте класс `Order` (заказ), который имеет следующие свойства и методы:

- Свойство `orderNumber` (номер заказа) - число, уникальный номер заказа.
- Свойство `products` (продукты) - массив, содержащий список продуктов в заказе.

- Метод `addProduct(product)` - принимает объект `product` и добавляет его в список продуктов заказа.
- Метод `getTotalPrice()` - возвращает общую стоимость заказа, основанную на ценах продуктов.

```
// Пример использования класса
class Product {
  constructor(name, price) {
    this.name = name;
    this.price = price;
  }
}

const order = new Order(12345);

const product1 = new Product("Phone", 500);
order.addProduct(product1);

const product2 = new Product("Headphones", 100);
order.addProduct(product2);

console.log(order.getTotalPrice()); // Вывод: 600
```