Прототипы

Курс: JavaScript, часть 2: прототипы и асинхронность

20 февраля 2018 г.

Оглавление

Глава 1

Прототипы

1.1. Прототипы

Сформулируем задачу: пусть у нас есть некоторый объект, который олицетворяет студента. Он записан в литеральной нотации и описывает характеристики студента, а также полезные для него действия. Этот объект содержит поле name, которое хранит имя студента и метод `getName`, который возвращает имя нашего студента.

```
var student = {
    name: 'Billy',
    type: 'human',
    getName: function () {
        return this.name;
    },
    sleep: function () {
        console.info('zzzZZZ...');
    }
};

student.getName();
// Billy
```

Объект студента сложно рассматривать в отрыве от объекта преподавателя. У преподавателя также есть ряд полей и методов, например поле `name`, которое хранит имя. Если мы посмотрим на два этих объекта внимательнее, мы уви-

дим, что в них очень много похожего: у каждого из них есть метод `getName`, который выполняет одинаковую работу. Таким образом, мы дублируем реализацию одного и того же метода в двух разных объектах, и это проблема.
К счастью, решение очень простое. Мы можем выделить общие части в отдельную конструкцию.
Назовем объединяющий объект person/личность. В итоге мы получим три несвязанных объекта: студента, преподавателя и личность.

```
var person = {
    type: 'human',
    getName: function () {
        return this.name;
    }
};
```

Так как мы забрали у наших объектов студента и преподавателя полезный метод `getName`, нам необходимо после нашего рефакторинга решить следующую задачу: научить студента пользоваться общим кодом, который мы вынесли в другой объект. Для решения этой задачи мы можем воспользоваться методом заимствования. Для этого мы можем позаимствовать метод `getName` у объекта `person` и вызвать его при помощи метода `call`, передав первым аргументом объект студента.

```
var student = {
    name: 'Billy',
};
var person = {
    getName: function () {
        return this.name;
    }
};
person.getName.call(student);
```

Нам хотелось бы вызывать метод `getName`, как и раньше, от лица студента. Можем ли мы связать два наших объекта студента и `person` таким образом, чтобы это было возможным?
Необходимо лишь в специальное внутреннее поле `[[Prototype]]` одного объекта записать ссылку на другой. Так, мы можем записать в это поле у объекта

`student` ссылку на объект `person` и получить желаемое поведение. Обратиться напрямую ко внутреннему полю, конечно, нельзя, но существует ряд способов, которые позволяют записать в него новое значение. Один из них — геттер и сеттер `_proto_`.

```
var student = {
    name: 'Billy',
    sleep: function () {},
    [[Prototype]]: <link to person>,
};
student['[[Prototype]]'] = person; //так не работает!
```

> Объект, на который указывает ссылка во внутреннем поле `[[Prototype]]`, называется **прототипом**.

1.2. Цепочки прототипов

Что происходит, когда мы пытаемся вызвать метод, которого нет у объекта, но он есть в прототипе? В этом случае интерпретатор переходит по ссылке, которая хранится во внутреннем поле `Prototype` и пробует найти этот метод в прототипе. В нашем случае мы вызываем метод `getName` у объекта student, но этого метода у этого объекта нет. Интерпретатор смотрит значения внутреннего поля `Prototype`, видит там ссылку на прототип — `person`, и переходит по этой ссылке, пробуя найти этот метод уже в прототипе. Там он этот метод находит и вызывает. Важно заметить, что `this` при исполнении этого метода будет ссылаться на объект `student`, так как мы этот метод вызываем от лица студента.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};
var person = {
    type: 'human',
    getName: function () {
        return this.name;
    }
};
```

Но что произойдет, если мы попытаемся вызвать метод, которого нет не только у объекта, но и в прототипе? Можно заметить, что у прототипа также есть внутреннее поле `Prototype`.

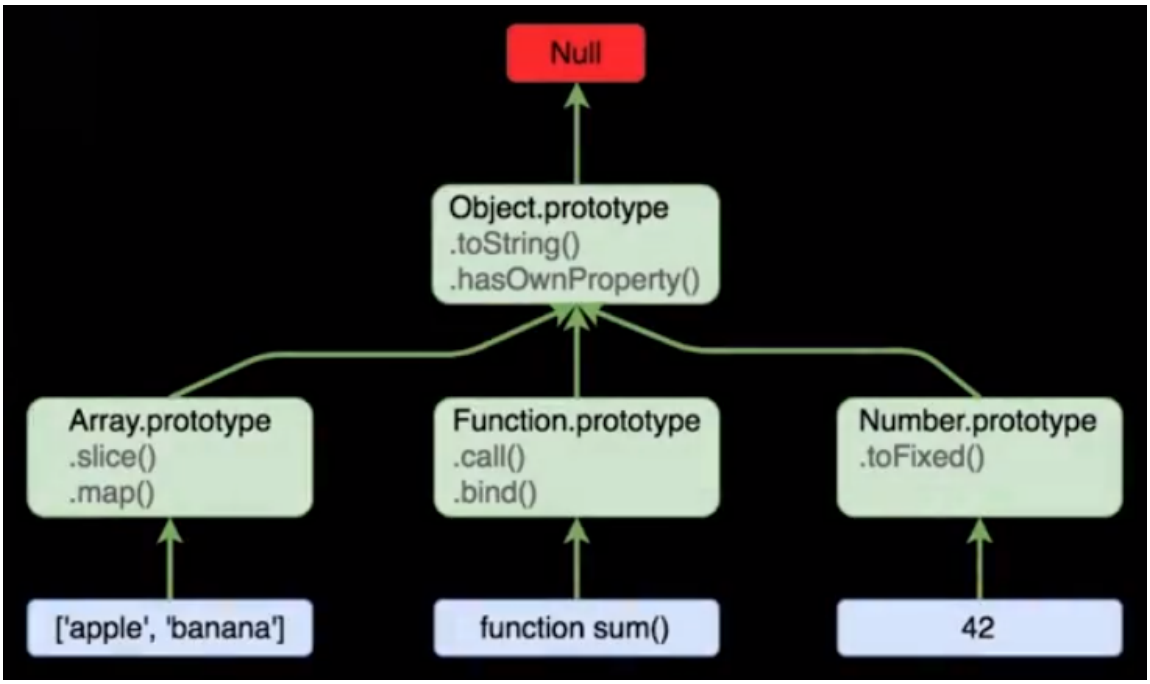
> Интерпретатор идет по выстроенной цепочке прототипов в поисках поля или метода до тех пор, пока не встретит значение `null` в специальном внутреннем поле `Prototype`. Если он прошел весь путь по цепочке, но так и не нашел искомого метода или поля, в этом случае он вернет `undefined`.

В нашем случае может показаться, что поиск остановится уже на объекте `person`, ведь мы специально не записывали никакую ссылку во внутреннее поле `Prototype`. Но любой объект уже имеет в качестве прототипа некоторый глобальный прототип — глобальный прототип для всех объектов. Он расположен в специальном поле `Prototype` функции `Object` и хранит в себе методы, полезные для всех объектов.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};
var person = {
    type: 'human',
    [[Prototype]]: <Object.prototype>
};
```

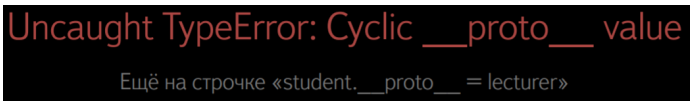
Мы выяснили, что в нашем случае поиск метода не остановится на объекте `person` и мы проследуем по цепочке прототипов дальше в глобальный прототип

для всех объектов. И уже там наш поиск остановится, так как внутреннее поле `Prototype` глобального прототипа для всех объектов имеет значение `null`. Помимо общего, глобального прототипа для всех объектов, существуют в языке более частные глобальные прототипы: для массивов, функций, и т.д. Каждый из этих прототипов по умолчанию в качестве прототипа имеет глобальный прототип для всех объектов. И для того, чтобы поиск метода или поля по цепочке прототипа всегда заканчивался, в этом глобальном прототипе в поле `Prototype`, во внутреннем поле, хранится значение `null`.



Давайте попробуем обмануть интерпретатор и в качестве прототипа для преподавателя выбрать студента, а в качестве прототипа для студента выбрать преподавателя. И попробуем вызвать заведомо несуществующий метод или поле. В данном случае может показаться, что поиск будет идти бесконечно, интерпретатор будет вечно ходить по созданному нами циклу в цепочке прототипов. Но интерпретатор позаботился о таком поведении и выбросит ошибку уже на этапе попытки создания такой цепочки.

```
1 var lecturer = { name:
  ↪  'Sergey'}
2 var student = { name:
  ↪  'Billy' }
3
4 lecturer.__proto__ =
  ↪  student;
5 student.__proto__ =
  ↪  lecturer;
6
7 console.info(lecturer.abrakadabra);
```



1.3. Способы установки прототипов

Есть три способа установки прототипа.

1.3.1. `__proto__`

Первый — это сеттер, геттер `__proto__`. Не идеальный метод:

- не является частью ECMAScript 5;
- он долгое время не являлся частью спецификации языка, и более того, поддерживался далеко не всеми платформами;
- появился он благодаря разработчикам браузеров, которые потихонечку внедряли эту возможность в свои продукты.

1.3.2. Метод `create`

Следующий способ установки прототипов — использование специального метода `create`, который в качестве параметра принимает в себя объект, который мы хотим видеть в качестве прототипа для нового объекта, который этот метод возвращает.

```
1 var student = Object.create(person)
```

Особенности способа:

- уже является частью ECMAScript 5;
- делает больше работы, чем простое присваивание ссылки;
- создаёт новые объекты и не может менять прототип существующих.

1.3.3. `setPrototypeOf`

Последний способ установки прототипа – специальный метод `setPrototypeOf`. Этот метод принимает уже два параметра: первый — исходный объект, а второй — объект, который мы хотим видеть в качестве прототипа для исходного.

```
var student = {
    name: 'Billy',
    sleep: function () {}
};

var person = {
    type: 'human',
    getName: function () {}
};

Object.setPrototypeOf(student, person);

student.getName();
// Billy
```

Особенности способа:

- появился только в ECMAScript 6
- близок к __proto__, но имеет особенность:
  - если мы попробуем присвоить через сеттер, геттер `__proto__` в качестве прототипа не объект, а число, то интерпретатор неявно проигнорирует это поведение; попробовав проделать тот же самый фокус с методом `setPrototypeOf`, интерпретатор поведет себя более явно и выбросит ошибку.

У метода `setPrototypeOf` есть парный метод `getPrototypeOf`. Этот метод возвращает ссылку на прототип. В отличие от `setPrototypeOf`, этот метод появился сравнительно давно в языке и позволяет нам проследовать по всей цепочке прототипов.

```
Object.getPrototypeOf(student) === person;
// true
Object.getPrototypeOf(person) === Object.prototype;
// true
Object.getPrototypeOf(Object.prototype) === null;
// true
```

1.4. Эффект затенения

Чтобы поменять значение какого-либо поля у объекта, нам достаточно выполнить простое присваивание. Но что, если мы попытаемся изменить поле, которого нет у объекта, но есть в его прототипе? Например, поле `type`.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
}

var person = {
    type: 'human',
    getName: function () {}
}

console.info(student.type); // human

student.type = 'robot';

console.info(student.type); // robot

console.info(person.type); // ???
```

Выполнив простое присваивание, мы добьемся желаемого. Может показаться,

что интерпретатор, не найдя это поле у студента, перейдет в прототип и поменяет значение уже там, но он оставит это поле в прототипе неприкосновенным. Вместо этого он создаст копию на стороне объекта, но уже с новым значением. Такой эффект называется **эффектом затенения свойства**.

```
console.info(person.type); // 'human'
```

Благодаря эффекту затенения мы можем переопределить методы, находящиеся в глобальном прототипе. Например, метод `toString`, который вызывается при приведении объекта к строке.

```
Object.prototype = {
    toString: function () {}
};
student.toString();
// [object Object]
console.info('Hello,' + student);
// Hello, [object Object]
```

1.5. Поля только для чтения в прототипах

Мы можем не просто установить поле, а задать ему некоторые характеристики, например, пометить это поле как изменяемое или неизменяемое.
Допустим, у нас есть объект студента с полем `name`, которое хранит его имя. Давайте добавим еще одно поле для этого объекта. Воспользуемся методом `defineProperty` и в качестве первого параметра передадим туда сам объект, в качестве второго параметра передадим название поля, а в качестве третьего — набор характеристик. Укажем значение поля, а также укажем, что это поле неизменяемое: зададим атрибуту `writable` значение `false`.

```
var student = { name: 'Billy' };
Object.defineProperty(student,'gender', {
    writable: false,
    value: 'male',
});
```

Если мы попытаемся перезаписать это начальное значение, то интерпретатор не даст нам этого сделать и сохранит исходное, причем сделает это неявно. Чтобы сделать поведение интерпретатора явным, нам необходимо переключиться в строгий режим интерпретации. Для этого понадобится добавить дополнительную директиву `use strict` в начало нашей программы. В этом случае при попытке перезаписать поле только для чтения интерпретатор бросит ошибку, в которой сообщит нам, что поле у объекта неизменяемое.

```
'use strict';
var student = { name: 'Billy' };
Object.defineProperty(student,
'gender'
, {
writable: false,
value: 'male'
});
```

Таким же образом работают неизменяемые поля в прототипах. Создадим новое поле в нашем прототипе person. Пусть это будет поле, которое хранит текущую планету, зададим ей начальное значение и атрибут `writable: false`. Мы увидим, что при попытке перезаписать это поле от лица исходного объекта, от лица студента, интерпретатор в строгом режиме также среагирует ошибкой, не даст нам этого сделать и скажет, что поле `planet` у объекта — неизменяемое.

```
Object.defineProperty(person,
'planet', {
    writable: false,
    value: 'Earth'
});

console.info(student.planet); // Earth

student.planet = 'Mars'; // TypeError: Cannot assign to read only
 ↪  property 'planet' of object
```

1.6. Сеттеры и геттеры в прототипах

Пусть у нас есть объект `student` с уже готовым полем, которое хранит имя студента. Мы хотим добавить для студента еще одно поле, которое будет хранить его возраст, такое, чтобы с ним было удобно работать.
Например, мы хотим передавать в это поле возраст в виде некоторой строки, но преобразовывать его внутри к числу. Для этого мы определяем сеттер и при помощи функции `parseInt` передаваемую строку, в которой содержится возраст, преобразуем к числу и сохраняем во внутреннее поле.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};

Object.defineProperty(student,'age', {
    set: function(age) { this._age = parseInt(age); },
    get: function() { return this._age; }
});

student.age = '20 лет';

console.info(student.age); // 20;
```

Далее мы определяем геттер, который позволяет получать уже готовое значение из этого внутреннего поля. Имеет смысл добавить поле возраста не конкретно к студенту, а в его прототип, в объект `person`. Для этого мы воспользуемся тем же самым методом `defineProperty`, но в качестве первого параметра передадим уже не студента, а прототип. Далее попробуем указать возраст для студента в виде строки и увидим, что все работает как надо.

```
var student = {
    [[Prototype]]: <person>
};
var person = {
    type: 'human'
};
Object.defineProperty(person,'age', {
    set: function(age) { this._age = parseInt(age); },
    get: function() { return this._age; }
});

student.age = '20 лет';

console.info(student.age); // 20;

student.hasOwnProperty(age); // false;
```

Здесь мы вспоминаем про эффект затенения: если мы попытаемся установить некоторое поле, которого нет у объекта, но есть в прототипе, поле в прототипе не будет изменено. Вместо этого интерпретатор создаст копию этого поля на объекте с новым значением. Но если поле в прототипе определено при помощи сеттера/геттера, данный эффект работать не будет, копия поля у объекта `student` не появится.

> Если поле в прототипе определено как геттер или сеттер, то эффект затенения **не** работает.

1.7. Неперечисляемые поля в прототипах

Мы можем пометить некоторые поля у объекта как перечисляемые (значение по умолчанию) или неперечисляемые. Если поля перечисляемые, то при помощи оператора `for...in` мы можем получить весь список полей объекта.
Более того, оператор `for...in` перечисляет не только поля самого объекта, но и поля связанного с ним прототипа. Допустим, если у нашего объекта есть прототип и в нём есть поля/методы, оператор `for...in` перечислит их наряду с полями объекта.
Это может быть нежелательным поведением, и, возможно, мы хотим перечислить именно собственные поля объекта, не затрагивая при этом поля в

прототипе. Для этого нам понадобится специальный метод `hasOwnProperty`, в качестве аргумента который принимает название поля. Данный метод просто отвечает на вопрос: принадлежит ли это поле объекту или нет. Добавив это условие в оператор `for...in`, мы можем вывести только собственные поля объекта.

```
var student = {
    name: 'Billy',
    age: 20,
    [[Prototype]]: <person>
};
var person = {
    type: 'human',
    getName: function () {}
};

for (var key in student)
    if (student.hasOwnProperty(key)) console.info(key);

// 'age', 'name'
```

Аналогично этой технике мы можем воспользоваться другим методом — методом `keys`, который хранится в функции `Object`. Для этого в этот метод мы передаём объект, а на выходе получаем массив из ключей полей объекта.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
}
var person = {
    type: 'human',
    getName: function () {}
}

var keys = Object.keys(student); // Получаем массив ключей

console.info(keys);

// ['name']
```

Чтобы добавить в объект неперечисляемое поле, воспользуемся методом `defineProperty`. Для этого передадим в него первым параметром сам объект, вторым параметром — название нового поля, а третьим параметром — характеристики. Укажем значение этого объекта. И с помощью специального атрибута укажем, что это поле неперечисляемое.

```
var student = { name: 'Billy' };

Object.defineProperty(student,'age', {
    enumerable: false,
    value: '20'
});

for (var key in student) console.info(key);
// 'name'

Object.keys(student);
// ['name']
```

В результате данное поле не будет участвовать в перечислениях, организуемых оператором `for...in` или методом `keys`.
Таким же образом мы можем задать неперечисляемое поле в прототипе, также воспользовавшись методом `defineProperty`. И данное поле также не будет участвовать в перечислениях.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};

var person = {
    type: 'human'
};

Object.defineProperty(person,'age', {
    enumerable: false
});

for (var key in student) console.info(key);
// 'name', 'type'
```

Важно заметить, что у глобальных прототипов для объектов или для массивов поля, обозначенные там, неперечисляемые. Мы не увидим их в попытке перечислить все поля конкретного объекта, несмотря на то, что в его прототипе может лежать глобальный прототип.

```
Object.prototype = {
    toString: function () {},
    [[Prototype]]: null
};
var person = {
    type: 'human',
    [[Prototype]]: <Object.prototype>
};

for (var key in person) console.info(key);

// 'type'
```

Конструкторы

Курс: JavaScript, часть 2: прототипы и асинхронность

20 февраля 2018 г.

Оглавление

Глава 1

Конструкторы

1.1. Конструкторы

Обычно в программе мы работаем не с одним конкретным объектом, а с целой коллекцией однотипных объектов. И нам необходимо уметь создавать объекты того же типа. Создание нового объекта такого же типа — достаточно громоздкая операция.
Можно вынести процесс создания новых объектов в конструктор.
Пусть это будет обычная функция — назовем ее `createStudent`, которая на вход принимает в качестве параметра имя студента, а на выходе отдает нам новый объект с заполненными полями и необходимыми методами.

```
1  function createStudent(name) {
2      return {
3          name: name,
4          sleep: function () {
5              console.info('zzzZZ ...');
6          }
7      };
8  }
9  var billy = createStudent('Billy');
10 var willy = createStudent('Willy');
```

Данное решение простое, но каждый раз при вызове конструктора мы будем создавать новую функцию, которая будет реализовывать метод `sleep`. Можно вынести его в прототип. Для этого создадим новый объект `studentProto`,

который будет являться прототипом для всех вновь создаваемых студентов, и перенесем туда наш метод `sleep`. После этого нам необходимо добавить туда вызов метода `setPrototypeOf`, который будет привязывать новых студентов к уже созданному нами прототипу. Каждый студент будет иметь доступ к методам, которые хранятся в прототипе для него.

```
var studentProto = {
    sleep: function () {
        console.info('zzzZZ ...');
    }
};
function createStudent(name) {
    var student = {
        name: name
    };
    Object.setPrototypeOf(student, studentProto);
    return student;
}
```

Мы можем воспользоваться уже готовым механизмом для создания конструкторов. Любая функция может быть конструктором, если мы вызовем ее при помощи специального оператора `new`. Функция `createStudent` может стать конструктором сама, но нам необходимо переписать ее реализацию и оставить только передаваемый аргумент, который будет хранить имя вновь создаваемого студента; всю остальную работу за нас сделает интерпретатор. Перед тем как исполнить код нашего конструктора, он создаст новый объект и присвоит его в переменную `this`. Далее мы заполним этот объект полями, и в конце интерпретатор за нас вернет объект, хранящийся по ссылке `this`. Можно сказать, что при вызове функции как конструктора с оператором `new` `this` внутри этой функции при исполнении будет указывать на вновь создаваемый объект.

```
var billy = new createStudent('Billy');

function createStudent(name) {
    // var this = {};
    this.name = name;
    // return this;
}
```

Такой код уже будет работать, но читается он не очень хорошо; переименуем нашу функцию просто в функцию `Student`.

```
function Student(name) {
    this.name = name;
}
var billy = new Student('Billy');
```

Функции-конструкторы принято именовать с заглавной буквы. Почему это важно?
Что произойдет, если мы попытаемся вызвать случайно, например, нашу функцию-конструктор как обычную функцию без оператора `new`? В этом случае переданное значение имени студента не будет записано в новый объект студента, а будет записано в глобальный объект в поле `name`, так как, вызывая функцию, `this` по умолчанию будет ссылаться на глобальный объект. Мы можем следовать соглашению и дополнительно включить строгий режим интерпретации, который защитит нас от такого поведения. В этом случае `this` будет иметь значение `undefined`, и мы не сможем присвоить в него никакие поля.

```
function Student(name) {
    this.name = name;
}
var billy = Student('Billy'); // Поле появится в глобальном объекте!

window.name === 'Billy'; // true

'use strict'; // TypeError: Cannot set property 'name' of undefined
```

Давайте попробуем вмешаться в работу конструктора, в работу интерпретатора и как-то поменять поведение конструктора. Например, мы захотим сами возвращать какой-то сконструированный объект. Можем ли мы это сделать? В данном случае интерпретатор нам полностью доверяет и вернет тот объект, который мы возвращаем при помощи оператора `return`.

```
function Student(name) {
    this.name = name;
    return {
        name: 'Muahahahahaha!'
    };
}
var billy = new Student('Billy');
console.info(billy.name); // Muahahahahaha
```

Но если мы попытаемся вернуть из конструктора какое-то примитивное значение — число, строку или `null`, в этом случае интерпретатор просто проигнорирует эту строку и будет работать как раньше: он будет возвращать вновь создаваемый объект.

```
function Student(name) {
this.name = name;
return null; // Evil mode on!
}
var billy = new Student('Billy');
console.info(billy.name);
// Billy
```

1.2. Конструкторы и прототипы

Допустим, у нас есть конструктор студентов и нам хотелось бы добавить в него несколько методов. Логично было бы хранить их в прототипе. Для того чтобы автоматически привязывать этот прототип для всех вновь создаваемых студентов, нам необходимо поместить его в хранилище. Оно есть у каждой функции конструктора в специальном поле prototype. Конструктор в момент исполнения выполняет дополнительный шаг: привязывает тот объект, который мы поместили в хранилище, в качестве прототипа для всех вновь создаваемых объектов; создавая новых студентов при помощи нашего конструктора, мы увидим, что у каждого из них внутреннее поле `prototype` будет ссылаться на тот объект, который мы ранее поместили в хранилище.

```
Student.prototype = {
    sleep: function () {}
};
function Student(name) {
    // var this = {};
    this.name = name;
    // Object.setPrototypeOf(this, Student.prototype);
    // return this;
}
```

Данное хранилище может вам напомнить другое хранилище, а именно то, которое расположено в специальном поле `prototype` функции `Object`. И более того, автоматически каждый создаваемый объект в JavaScript имеет в качестве прототипа объект этого хранилища. Изначально нам было не очень очевидно, каким же образом осуществлялась данная привязка, ведь мы создавали объекты при помощи литеральной конструкции, а не конструктора и оператора `new`. Но под капотом интерпретатор вызывает тот же самый конструктор `Object` оператором `new`.
Давайте подробнее поговорим про специальное поле `.prototype`:

- есть у каждой функции;
- хранит объект;
- имеет смысл только при вызове функции как конструктора;
- имеет вложенное поле `.constructor` (неперечисляемое, хранит ссылку на саму функцию).

Обращаясь к полю `.constructor`, мы можем получить доступ к конструктору, т.е можем, например, выяснить имя конструктора конкретного объекта. Например, если мы сконструировали нового студента `Billy` на основе конструктора Student, мы можем обратиться к этому полю у `Billy`. У `Billy` этого поля нет, но оно есть в прототипе, который, как мы знаем, изначально хранится в хранилище Student.`prototype`. И так как данное поле хранит ссылку на функцию, мы можем посмотреть имя этой функции, обратившись к полю name.

```
function Student(name) {
    this.name = name;
}

Student.prototype.constructor === Student; // true
var billy = new Student('Billy');
console.info(billy.constructor.name); // Student
```

Важно помнить о поле `.constructor`, так как мы очень легко можем его перезаписать. В нашем случае мы это и сделали. Мы просто поместили в поле .prototype, в это хранилище, новый объект, тем самым уничтожив поле `.constructor`.

```
function Student(name) {
    this.name = name;
}
Student.prototype = {
    sleep: function () {}
};
```

Чтобы этого не произошло, достаточно не перезаписывать тот объект, который хранится изначально в этом хранилище, а дополнять его новыми методами. Таким образом, мы получим на основе этого конструктора новые объекты. Они будут иметь доступ как к методам из прототипа, так и к специальному полю `.constructor`.

```
function Student(name) {
    this.name = name;
}

Student.prototype.sleep = function () {
    console.info('zzzZZ ...');
}

var Billy = new Student('Billy');
billy.sleep(); // zzzZZ ...
billy.constructor === Student; // true
```

```
function Student(name) {
    this.name = name;
}
Student.prototype.sleep = function () {};
```

```
function Person() {
    this.type = 'human';
}
Person.prototype.getName = function () {
    return this.name;
}
```

1.3. Конструкторы и цепочки прототипов

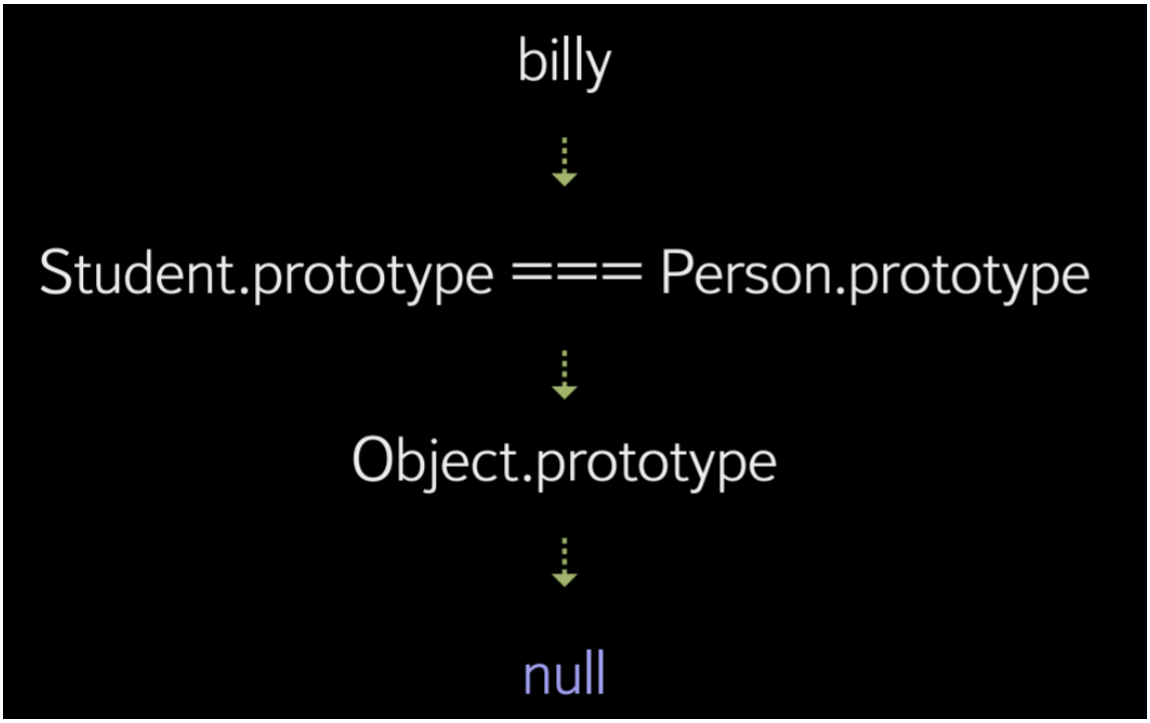
У нас есть некоторый конструктор студентов, который принимает на вход имя студента и записывает его в поле name. А также хранилище, в поле .prototype нашего конструктора мы поместили объект, который хотим видеть в качестве прототипа для всех вновь создаваемых студентов. Там сейчас один метод – sleep.
Допустим, у нас есть более абстрактный конструктор, который также имеет хранилище и в котором расположен другой метод – getName.
В данном случае нам бы не хотелось дублировать реализацию этого метода в хранилище конструктора студентов. Для этого мы можем пойти самым простым путем и хранилище конструктора студентов сделать на основе хранилища конструктора Person: присваиваем в специальное поле prototype ссылку на хранилище конструктора Person. Далее, мы можем расширить наше хранилище для студентов более специфичным для них методом – методом sleep. Итак, мы можем создавать теперь новых студентов, которые будут иметь доступ к методу sleep из своего хранилища и к методу getName, которое изна-

```
function Student(name) {
this.name = name;
}
Student.prototype = Person.prototype;
Student.prototype.sleep = function () {};
```

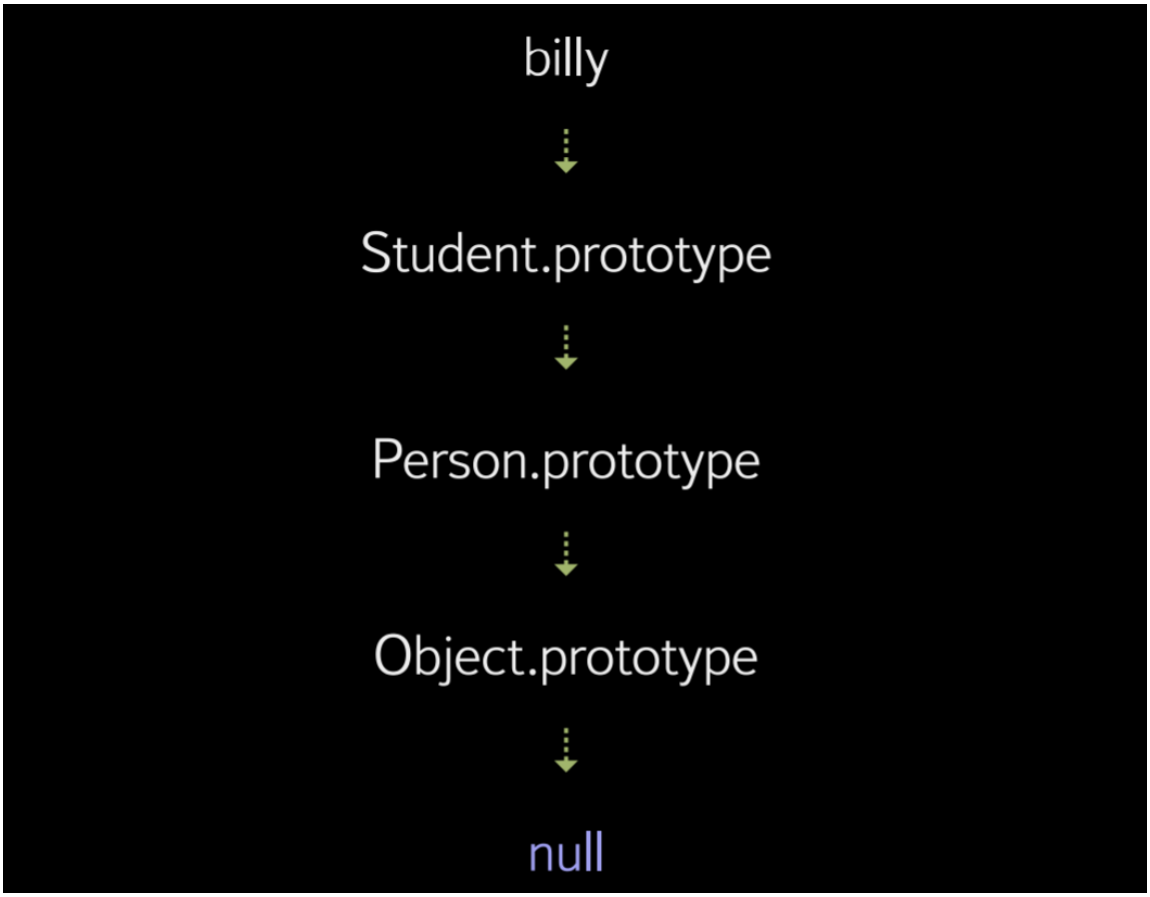
чально было в хранилище конструктора `Person`.
Данный способ имеет подводный камень. Если мы в своей программе попробуем использовать объекты другого типа, создадим для них конструктор и в хранилище этого конструктора запишем ссылку на то же самое хранилище, которое связано с конструктором `Person`, безусловно, наш новый объект – преподаватель – получит доступ к методам из этого хранилища `Person`. Но мы с удивлением обнаружим, что преподаватель получит доступ и к методам, которые мы ранее определили только для студентов, например к методу `sleep`.

```
function Student(name) {
    this.name = name;
}
Student.prototype = Person.prototype;
Student.prototype.sleep = function () {};

function Lecturer(name) {
    this.name = name;
}
Lecturer.prototype = Person.prototype;

var sergey = new Lecturer('Sergey');

sergey.sleep(); // zzzZZ ...
```

Это происходит потому, что все три этих хранилища хранят сейчас ссылки на один и тот же объект. Наша цепочка прототипов выглядит примерно так.



Это нежелательное поведение и нам бы хотелось, чтобы цепочка выглядела примерно так.



1.3.1. Метод `create`

Решить эту проблему нам поможет специальный метод `create`. Вместо обычного присваивания мы в хранилище для конструктора студентов будем записывать объект, который нам этот метод возвращает. Такой же трюк мы можем проделать и с конструктором преподавателя. Попробуем теперь создать нового преподавателя и вызвать у него метод, который мы определили только для студентов. Благодаря методу `create`, мы увидим, что преподаватели не будут иметь доступ к методам, которые мы определили только для студентов.

```
function Student(name) {
    this.name = name;
}

Student.prototype = Object.create(Person.prototype);
Student.prototype.sleep = function () {};


function Lecturer(name) {
    this.name = name;
}
Lecturer.prototype = Object.create(Person.prototype);

var sergey = new Lecturer('Sergey');
sergey.sleep(); //TypeError: sergey.sleep is not a function
```

Попробуем разобраться, как работает метод `create`. Метод создает пустой объект, прототипом которого становится объект, переданный первым аргументом. Есть прототип для фруктов, которые хранит единственное поле. Оно говорит нам о том, что все фрукты полезные. И на основе этого прототипа фруктов мы будем создавать новые фрукты при помощи метода `create`, передавая в качестве аргумента наш прототип. Так мы можем создать яблоко и проверить, что оно действительно полезное, несмотря на то, что этого поля у самого яблока нет. Оно есть у его прототипа.

```
var fruitProto = {
isUsefull: true
}
var apple = Object.create(fruitProto);
apple.isUsefull; // true
```

Внутри метод `create` устроен достаточно просто. Он создает простейшие конструкторы из возможных, а именно: пустую функцию. Далее, в хранилище этого конструктора он записывает ссылку на тот объект, который мы передаем в качестве первого аргумента, тот объект, который хотим видеть в качестве прототипа для всех создаваемых объектов. Далее, он при помощи этого конструктора создает новый объект и возвращает его. Таким образом, все вновь

создаваемые объекты будут иметь в качестве прототипа тот объект, который мы передаем первым аргументом.

```
var apple = Object.create(fruitProto);

Object.create = function(prototype) {
    // Простейший конструктор пустых объектов
    function EmptyFunction() {};
    EmptyFunction.prototype = prototype;
    return new EmptyFunction();
};
```

В метод `create` мы можем передавать не только объекты, но и, например, значение `null`. В этом случае мы создадим объект, в качестве прототипа которого не будет выступать ни один из объектов, даже глобальный прототип для всех объектов. И мы не получим доступ к методам из этого глобального прототипа. Метод `create` помогает нам связать два хранилища разных конструкторов так, чтобы они не ссылались на один и тот же объект. И хранилище для конструктора студентов будет представлять из себя отдельный объект, но во внутреннем поле `prototype` которого лежит ссылка на другое хранилище – хранилище конструктора `Person`. Далее мы можем расширить хранилище для студентов специфичными для них методами.

```
function Student(name) {
    this.name = name;
}
Student.prototype = Object.create(Person.prototype);
Student.prototype.sleep = function () {};
```

И здесь мы допустили ту же самую ошибку, что и ранее. Мы полностью перезаписали хранилище студентов и забыли о поле конструктора.
Давайте его вернем. Достаточно просто присвоить в него ссылку на функцию.

```
function Student(name) {
    this.name = name;
}
Student.prototype = Object.create(Person.prototype);
Student.prototype.sleep = function () {};

Student.prototype.constructor = Student;
```

Итоговое решение нашей задачи выглядит примерно так.

```
function Person() {
    this.type = 'human';
}

Person.prototype.getName = function () {
    return this.name;
};

function Student(name) {
    this.name = name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.sleep = function () {};

Student.prototype.constructor = Student;

var billy = new Student('Billy');
```

1.4. Инспектирование связей между объектами, конструкторами и прототипами

1.4.1. `getPrototypeOf`

Первый способ — это метод `getPrototypeOf`. На вход он принимает в себя объект, а на выходе дает ссылку на прототип для этого объекта. В данном случае

у нас есть объект студента, в качестве прототипа для которого выступает объект `person`. Вызвав метод `getPrototypeOf` и передав туда ссылку на объект студента, мы на выходе получим на объект `person`.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
}

var person = {
    type: 'human',
    getName: function () {}
}

Object.getPrototypeOf(student) === person; // true
```

1.4.2. `isPrototypeOf`

Следующий способ инспектирования связей между объектами и прототипом предлагает нам метод `isPrototypeOf`. Допустим, у нас есть некий конструктор студентов, который на вход принимает имя студента и сохраняет его в поле `name` для каждого студента; хотим к каждому студенту привязать некоторый прототип. Мы помещаем этот прототип в поле `prototype`, в хранилище, и делаем его объектом на основе другого хранилища — конструктора `person` при помощи метода `create`. Далее, мы в наше хранилище добавляем специфичный для студентов метод `sleep` и восстанавливаем конструктор. Попробуем теперь создать нашего студента и воспользоваться для проверки связи между созданным студентом и его прототипом методом is`prototype`Of. Данный метод отвечает на вопрос: «Является ли объект прототипом для того объекта, который мы передаем в качестве аргументов?»

```
function Student(name) {
    this.name = name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.sleep = function () {};

Student.prototype.constructor = Student;

var billy = new Student('Billy');

Student.prototype.isPrototypeOf(billy); // true
```

Более того, данный метод позволяет инспектировать не только прямую связь, но и связь конечного объекта с одним из прототипов цепочки. Таким образом, он даст утвердительный ответ и на вопросы: «Является ли объект, который лежит в хранилище конструктора `person`, прототипом для `Billy`?» и «Является ли глобальный прототип для всех объектов прототипом для `Billy`?»

```
Person.prototype.isPrototypeOf(billy); // true

Object.prototype.isPrototypeOf(billy); // true
```

1.4.3. `instanceof`

Следующий способ инспектирования связей между объектами и конструкторами — специальный оператор `instanceof`. Он позволяет ответить на вопрос: «Является ли объект объектом определенного конструктора?» В данном случае мы создали нового студента `Billy` на основе конструктора Student и проверяем, является ли `Billy` объектом этого конструктора.

```
function Student(name) {
    this.name = name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.sleep = function () {};

Student.prototype.constructor = Student;

var billy = new Student('Billy');

billy instanceof Student; // true
```

Но этот оператор работает несколько сложнее, и он ответит, что `Billy` является и объектом конструктора `Person`, хотя напрямую это не так.

```
billy instanceof Person; // true

billy instanceof Object; // true
```

Но можно заметить, что хранилище, связанное с конструктором `Person`, лежит в цепочке прототипов до `Billy`. Если мы немножечко переформулируем то, как работает этот оператор, все встанет на свои места. Можно сказать, что он отвечает на вопрос: «Является ли `Billy` студентом?» или: «Является ли `Billy` личностью?» Более того, `Billy`, конечно же, является объектом — тоже правда.

Давайте разберем, как работает оператор `instanceof`. Например, как он проверяет, связаны ли между собой объект `Billy` с конструктором `Person`. Вначале он проверяет, является ли прототипом для `Billy` объект, который хранится в хранилище этого конструктора. И это не так. Тогда он проверяет следующую гипотезу: «А, может быть, прототипа у `Billy` совсем нет, и там хранится значение `null`?» Эта гипотеза также не подтверждается, и тогда он идет по цепочке прототипов. Он проверяет, является ли прототип прототипа `Billy` объектом, который хранится в хранилище конструктора `Person`. На этот раз гипотеза подтверждается, и в этом случае данный оператор отвечает нам `true`.

```
billy instanceof Person;
billy.__proto__ === Person.prototype;
// false -> Может, там null?
billy.__proto__ === null;
// false -> Идём дальше по цепочке
billy.__proto__.__proto__ === Person.prototype;
// true -> Возвращаем true
```

Если мы попробуем проинспектировать объект с самой короткой цепочкой прототипов, оператор `instanceof` возвращает нам `false`. Мы создаем объект и проверяем, что этот объект — это действительно объект, но оператор `instanceof` возвращает нам `false`. Если мы еще раз посмотрим, как работает этот оператор, все встанет на свои места. Итак, в первую очередь, он проверяет, является ли прототипом для нашего одинокого объекта глобальный прототип для всех объектов. Это не так. Тогда он проверяет: «Возможно, у нашего одинокого объекта совсем нет прототипа, и во внутреннем поле `prototype` лежит значение `null`?» В этом случае оператор `instanceof` при положительном подтверждении такой гипотезы возвращает нам `false`.

```
var foreverAlone = Object.create(null);

foreverAlone instanceof Object; // false

Object.create(null).__proto__ === Object.prototype;
// false -> Может, там null?

Object.create(null).__proto__ === null;
// true -> Так и есть, возвращаем false!
```

1.5. Решение проблемы дублирования кода в конструкторах

У нас есть конструктор студентов, конструктор преподавателей и конструктор личностей. Если мы посмотрим на конструкторы преподавателей и студентов, мы увидим, что они похожи и выполняют одинаковую работу: принимают в

качестве аргумента имя студента или преподавателя и сохраняют его в поле `name`.

```
function Student(name) {
    this.name = name;
}

function Lecturer(name) {
    this.name = name;
}

function Person() {
    this.type = 'human';
}
```

Нам бы хотелось избежать этого дублирования. Проще всего — вынести этот общий код в отдельный конструктор `Person`. Перенесем туда строчку, которая сохраняет имя студента или преподавателя в конструктор `Person`.

```
function Student(name) {
    this.name = name;
}

function Lecturer(name) {
    this.name = name;
}

function Person() {
    this.type = 'human';
}
```

Если мы сейчас попытаемся создать нового студента и передадим туда имя, мы не получим желаемого результата, так как мы забрали этот код из конструктора для студентов и переместили его в конструктор для личностей. Поэтому нам необходимо немножечко изменить реализацию конструктора студентов и добавить туда вызов конструктора личностей. Мы можем это сделать при помощи метода `call`. Вызываем наш конструктор личностей при помощи этого

метода и в качестве контекста передаем туда `this`, а в качестве второго аргумента передаем имя создаваемого нами студента.

```
function Person(name) {
    this.type = 'human';
    this.name = name;
}

function Student(name) {
    // this ссылается на новый объект студента
    Person.call(this, name);
}

var billy = new Student('Billy');

console.info(billy.name); // Billy
```

Благодаря тому, что в конструкторе `this` будет ссылаться на новый создаваемый объект, в нашем случае — на нового студента, мы вызовем конструктор `Person` с контекстом в виде этого студента, и таким образом передаваемое имя мы положим в поле name именно для студента. Все будет работать.

1.6. Вызов затеняемого метода в затеняющем

Допустим, у нас есть некоторый конструктор `Person`, конструктор личностей, который на вход принимает имя этих личностей и кладет в поле name. Также мы определяем прототип для всех личностей с методом `getName`, который возвращает это имя. Положим этот прототип в хранилище конструктора `Person`.

```
function Person(name) {
    this.name = name;
}

Person.prototype.getName = function () {
    return this.name;
}
```

Далее мы захотим использовать в нашей программе объекты другого типа и создадим конструктор для них. Прототипом для них мы сделаем объект на основе прототипа для личностей, но захотим изменить метод `getName`. Нам не хочется дублировать тот код, который есть в методе `getName` прототипа личностей, а лишь дополнить и добавлять к нему некую строку.
Вначале нам может показаться, что мы можем вызвать просто метод `getName` из прототипа `Person` внутри метода `getName` прототипа для студентов. Но в этом случае произойдет рекурсивный вызов того же самого метода. И интерпретатор нам скажет, что количество вызовов заполнило весь стек.

```
1  Student.prototype = Object.create(Person.prototype);
2
3  Student.prototype.getName = function () {
4      return 'Student ' + this.getName();
5  };
6
7  var billy = new Student('Billy');
8
9  billy.getName(); // RangeError: Maximum call stack size exceeded
```

Когда мы вызываем метод `getName` у созданного нами студента, например, `Billy`, внутри этого метода `this` будет ссылаться, собственно, на этого самого студента. Так как метода `getName` у самого `Billy` нет, он пойдет искать его в прототипе и найдет метод в прототипе, который хранится в хранилище конструктора `Student` в поле `Student.prototype`. Фактически метод будет вызывать сам себя.
Каким образом мы можем решить эту проблему? Самое простое — использовать другое название метода вместо эффекта затенения. В этом случае все будет работать. Мы будем вызывать метод с другим названием, например, getStudentName, заходить внутрь этого метода, `this` внутри него будет попрежнему ссылаться на объект `Student`, то есть на `Billy`, попробуем найти у `Billy` метод `getName`, не найдем его там, пройдем по цепочке прототипов в объект, который хранится в хранилище конструктора студентов Student.`prototype`. И там мы этого метода не найдем, проследуем дальше по цепочке прототипов и уже перейдем в хранилище, которое хранится в конструкторе `Person`. Там мы этот метод находим, спокойно его вызываем, возвращаем имя студента, добавляем к нему наш необходимый префикс, и все работает, как нужно.

```
function Person(name) {
    this.name = name;
}

Person.prototype.getName = function () {
    return this.name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.getStudentName = function () {
    return 'Student ' + this.getName();
};

var billy = new Student('Billy');

billy.getStudentName();
```

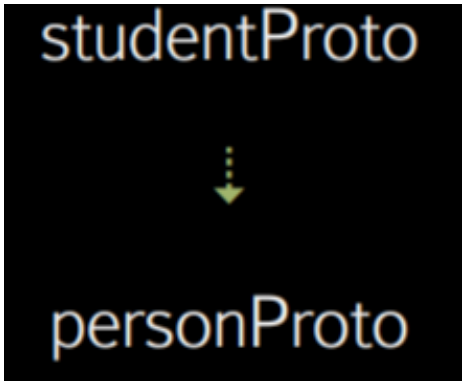
Более элегантным способом будет использование метода `call`. Мы можем напрямую в затеняющем методе вызывать затеняемый при помощи этого метода `call`, но передавать туда текущий контекст. А текущий контекст будет ссылаться на создаваемого объекта конструктора студентов, а именно на студента. Таким образом мы вызываем затеняемый метод от лица этого самого студента, получаем имя этого студента и добавляем к нему префикс `student`.

```
function Person(name) {
    this.type = 'human';
    this.name = name;
}

Person.prototype.getName = function () {
    return this.name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.getName = function () {
    return 'Student ' + Person.prototype.getName.call(this);
};
```

1.7. Сравнение трёх подходов к конструированию объектов: функции-конструкторы, метод `create`, «Классы»

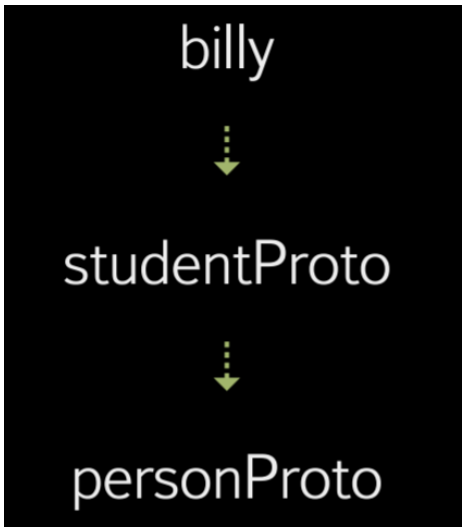
Для конструирования объектов и построения связей между ними достаточно лишь использовать только метод `create`.
Давайте снова попробуем создать конструктор для студентов. Для этого нам вновь понадобится ряд прототипов, выстроенных в цепочку. На этот раз мы воспользуемся только методом `create` и обычными объектами, так как любой прототип по факту — это обычный объект. Итак, для начала нам понадобится прототип личности. Создадим простой объект `personProto` и добавим туда метод `getName`, который будет возвращать имя нашей личности. На основе этого прототипа создадим более специфичный прототип уже для студентов. При помощи метода `create` мы свяжем два этих прототипа в цепочку. Далее мы можем расширить наш прототип для студентов уже специфичными для студентов методами, например, методом `sleep`.

```
var personProto = {
    getName: function () {
        return this.name;
    }
};
var studentProto =
 ↪  Object.create(personProto);

studentProto.sleep = function () {};
```



Далее на основе прототипа для студентов мы уже будем создавать студентов. Для этого снова воспользуемся методом `create`. Таким образом, благодаря этому методу мы встроим нашего студента в цепочку прототипов. И далее всё, что нам останется, это только дополнить нашего студента полями, которые необходимы каждому конкретному студенту с конкретными значениями. В данном случае мы присваиваем нашему студенту имя `Billy`.

```
var billy =
 ↪  Object.create(studentProto);

billy.name = 'Billy';
```



Благодаря такому подходу нам понадобилось значительно меньше строчек кода, чем в классическом, чтобы научить нашу программу создавать новые объекты студентов. Но и здесь мы можем произвести улучшение. Для этого нам понадобится ещё одна возможность метода `create`, а именно — данный метод принимает не один аргумент, а два. В качестве второго аргумента вы можете

передать необходимые поля с их начальными значениями и характеристиками, которые вы хотите видеть при создании объекта в итоговом объекте.

```
var apple = Object.create(fruit, {
    shape: { value: 'round', writable: false },
    color: { value: 'Green' },
    amount: { writable: true }
});

apple.amount = 'half';
```

Посмотрим, каким образом мы можем использовать данную возможность. Чтобы создавать наших студентов в одну строку, нам понадобится дополнительная функция-помощник, назовём её `create` и положим в наш прототип для студентов. Иногда такие функции называют фабриками — фабриками объектов. Данная функция будет на вход принимать в себя имя студента, а внутри себя вызывать метод `Object.create`, в качестве контекста передавать `this` и в качестве второго аргумента передавать набор полей, которые мы хотим видеть у студента, а именно: мы хотим у него видеть поле `name`, и мы сразу заполняем его тем, что нам приходит в фабрику. Таким образом мы можем создавать новых студентов в одну строчку, как и при классическом подходе при помощи оператора `new`. В данном случае мы просто вызываем наш метод `create`, который лежит в прототипе студента, передаём туда имя и получаем новый объект, нового студента с этим именем.

```
var personProto = {};

personProto.getName = function () { return this.name; }

var studentProto = Object.create(personProto);

studentProto.sleep = function () {};

studentProto.create = function (name) {
    return Object.create(this, {
        name: { value: name }
    });
}

var billy = studentProto.create('Billy');
```

Единого мнения, какой из этих подходов лучше, у разработчиков нет.
Также есть новая версия спецификаций, которая вводит новую синтаксическую конструкцию, а именно «классы». И вместо функций-конструкторов мы можем определить класс и на основе этого класса создавать новые объекты.

```
function Student(name) {
    this.name = name;
}

Student.prototype.getName = function () {
    return this.name
};

class Student {
    constructor(name) {
        this.name = name;
    }

    getName() {
        return this.name;
    }
}
```

Классы - это не более, чем обычные функции-конструкторы. У них также есть специальное поле `prototype` — хранилище для прототипов, для всех вновь создаваемых этим классом объектов. Также если мы посмотрим тип классов, то мы увидим, что это обычная функция.

```
class Student {
// ...
}

var billy = new Student('Billy');

billy.getName(); // Billy

Student.prototype.isPrototypeOf(billy); // true

typeof Student; // function
```

Таким образом, дискуссия трансформировалась в другую: что выбрать, классы или метод `Object.create`? Наиболее полно её раскрыл в статье разработчик Eric Elliott, и я рекомендую вам всем ознакомиться с ней.

Конструкторы

Курс: JavaScript, часть 2: прототипы и асинхронность

20 февраля 2018 г.

Оглавление

Глава 1

Конструкторы

1.1. Конструкторы

Обычно в программе мы работаем не с одним конкретным объектом, а с целой коллекцией однотипных объектов. И нам необходимо уметь создавать объекты того же типа. Создание нового объекта такого же типа — достаточно громоздкая операция.
Можно вынести процесс создания новых объектов в конструктор.
Пусть это будет обычная функция — назовем ее `createStudent`, которая на вход принимает в качестве параметра имя студента, а на выходе отдает нам новый объект с заполненными полями и необходимыми методами.

```
1  function createStudent(name) {
2      return {
3          name: name,
4          sleep: function () {
5              console.info('zzzZZ ...');
6          }
7      };
8  }
9  var billy = createStudent('Billy');
10 var willy = createStudent('Willy');
```

Данное решение простое, но каждый раз при вызове конструктора мы будем создавать новую функцию, которая будет реализовывать метод `sleep`. Можно вынести его в прототип. Для этого создадим новый объект `studentProto`,

который будет являться прототипом для всех вновь создаваемых студентов, и перенесем туда наш метод `sleep`. После этого нам необходимо добавить туда вызов метода `setPrototypeOf`, который будет привязывать новых студентов к уже созданному нами прототипу. Каждый студент будет иметь доступ к методам, которые хранятся в прототипе для него.

```
var studentProto = {
    sleep: function () {
        console.info('zzzZZ ...');
    }
};
function createStudent(name) {
    var student = {
        name: name
    };
    Object.setPrototypeOf(student, studentProto);
    return student;
}
```

Мы можем воспользоваться уже готовым механизмом для создания конструкторов. Любая функция может быть конструктором, если мы вызовем ее при помощи специального оператора `new`. Функция `createStudent` может стать конструктором сама, но нам необходимо переписать ее реализацию и оставить только передаваемый аргумент, который будет хранить имя вновь создаваемого студента; всю остальную работу за нас сделает интерпретатор. Перед тем как исполнить код нашего конструктора, он создаст новый объект и присвоит его в переменную `this`. Далее мы заполним этот объект полями, и в конце интерпретатор за нас вернет объект, хранящийся по ссылке `this`. Можно сказать, что при вызове функции как конструктора с оператором `new` `this` внутри этой функции при исполнении будет указывать на вновь создаваемый объект.

```
var billy = new createStudent('Billy');

function createStudent(name) {
    // var this = {};
    this.name = name;
    // return this;
}
```

Такой код уже будет работать, но читается он не очень хорошо; переименуем нашу функцию просто в функцию `Student`.

```
function Student(name) {
    this.name = name;
}
var billy = new Student('Billy');
```

Функции-конструкторы принято именовать с заглавной буквы. Почему это важно?
Что произойдет, если мы попытаемся вызвать случайно, например, нашу функцию-конструктор как обычную функцию без оператора `new`? В этом случае переданное значение имени студента не будет записано в новый объект студента, а будет записано в глобальный объект в поле `name`, так как, вызывая функцию, `this` по умолчанию будет ссылаться на глобальный объект. Мы можем следовать соглашению и дополнительно включить строгий режим интерпретации, который защитит нас от такого поведения. В этом случае `this` будет иметь значение `undefined`, и мы не сможем присвоить в него никакие поля.

```
function Student(name) {
    this.name = name;
}
var billy = Student('Billy'); // Поле появится в глобальном объекте!

window.name === 'Billy'; // true

'use strict'; // TypeError: Cannot set property 'name' of undefined
```

Давайте попробуем вмешаться в работу конструктора, в работу интерпретатора и как-то поменять поведение конструктора. Например, мы захотим сами возвращать какой-то сконструированный объект. Можем ли мы это сделать? В данном случае интерпретатор нам полностью доверяет и вернет тот объект, который мы возвращаем при помощи оператора `return`.

```
function Student(name) {
    this.name = name;
    return {
        name: 'Muahahahahaha!'
    };
}
var billy = new Student('Billy');
console.info(billy.name); // Muahahahahaha
```

Но если мы попытаемся вернуть из конструктора какое-то примитивное значение — число, строку или `null`, в этом случае интерпретатор просто проигнорирует эту строку и будет работать как раньше: он будет возвращать вновь создаваемый объект.

```
function Student(name) {
this.name = name;
return null; // Evil mode on!
}
var billy = new Student('Billy');
console.info(billy.name);
// Billy
```

1.2. Конструкторы и прототипы

Допустим, у нас есть конструктор студентов и нам хотелось бы добавить в него несколько методов. Логично было бы хранить их в прототипе. Для того чтобы автоматически привязывать этот прототип для всех вновь создаваемых студентов, нам необходимо поместить его в хранилище. Оно есть у каждой функции конструктора в специальном поле prototype. Конструктор в момент исполнения выполняет дополнительный шаг: привязывает тот объект, который мы поместили в хранилище, в качестве прототипа для всех вновь создаваемых объектов; создавая новых студентов при помощи нашего конструктора, мы увидим, что у каждого из них внутреннее поле `prototype` будет ссылаться на тот объект, который мы ранее поместили в хранилище.

```
Student.prototype = {
    sleep: function () {}
};
function Student(name) {
    // var this = {};
    this.name = name;
    // Object.setPrototypeOf(this, Student.prototype);
    // return this;
}
```

Данное хранилище может вам напомнить другое хранилище, а именно то, которое расположено в специальном поле `prototype` функции `Object`. И более того, автоматически каждый создаваемый объект в JavaScript имеет в качестве прототипа объект этого хранилища. Изначально нам было не очень очевидно, каким же образом осуществлялась данная привязка, ведь мы создавали объекты при помощи литеральной конструкции, а не конструктора и оператора `new`. Но под капотом интерпретатор вызывает тот же самый конструктор `Object` оператором `new`.
Давайте подробнее поговорим про специальное поле `.prototype`:

- есть у каждой функции;
- хранит объект;
- имеет смысл только при вызове функции как конструктора;
- имеет вложенное поле `.constructor` (неперечисляемое, хранит ссылку на саму функцию).

Обращаясь к полю `.constructor`, мы можем получить доступ к конструктору, т.е можем, например, выяснить имя конструктора конкретного объекта. Например, если мы сконструировали нового студента `Billy` на основе конструктора Student, мы можем обратиться к этому полю у `Billy`. У `Billy` этого поля нет, но оно есть в прототипе, который, как мы знаем, изначально хранится в хранилище Student.`prototype`. И так как данное поле хранит ссылку на функцию, мы можем посмотреть имя этой функции, обратившись к полю name.

```
function Student(name) {
    this.name = name;
}

Student.prototype.constructor === Student; // true
var billy = new Student('Billy');
console.info(billy.constructor.name); // Student
```

Важно помнить о поле `.constructor`, так как мы очень легко можем его перезаписать. В нашем случае мы это и сделали. Мы просто поместили в поле .prototype, в это хранилище, новый объект, тем самым уничтожив поле `.constructor`.

```
function Student(name) {
    this.name = name;
}
Student.prototype = {
    sleep: function () {}
};
```

Чтобы этого не произошло, достаточно не перезаписывать тот объект, который хранится изначально в этом хранилище, а дополнять его новыми методами. Таким образом, мы получим на основе этого конструктора новые объекты. Они будут иметь доступ как к методам из прототипа, так и к специальному полю `.constructor`.

```
function Student(name) {
    this.name = name;
}

Student.prototype.sleep = function () {
    console.info('zzzZZ ...');
}

var Billy = new Student('Billy');
billy.sleep(); // zzzZZ ...
billy.constructor === Student; // true
```

```
function Student(name) {
    this.name = name;
}
Student.prototype.sleep = function () {};
```

```
function Person() {
    this.type = 'human';
}
Person.prototype.getName = function () {
    return this.name;
}
```

1.3. Конструкторы и цепочки прототипов

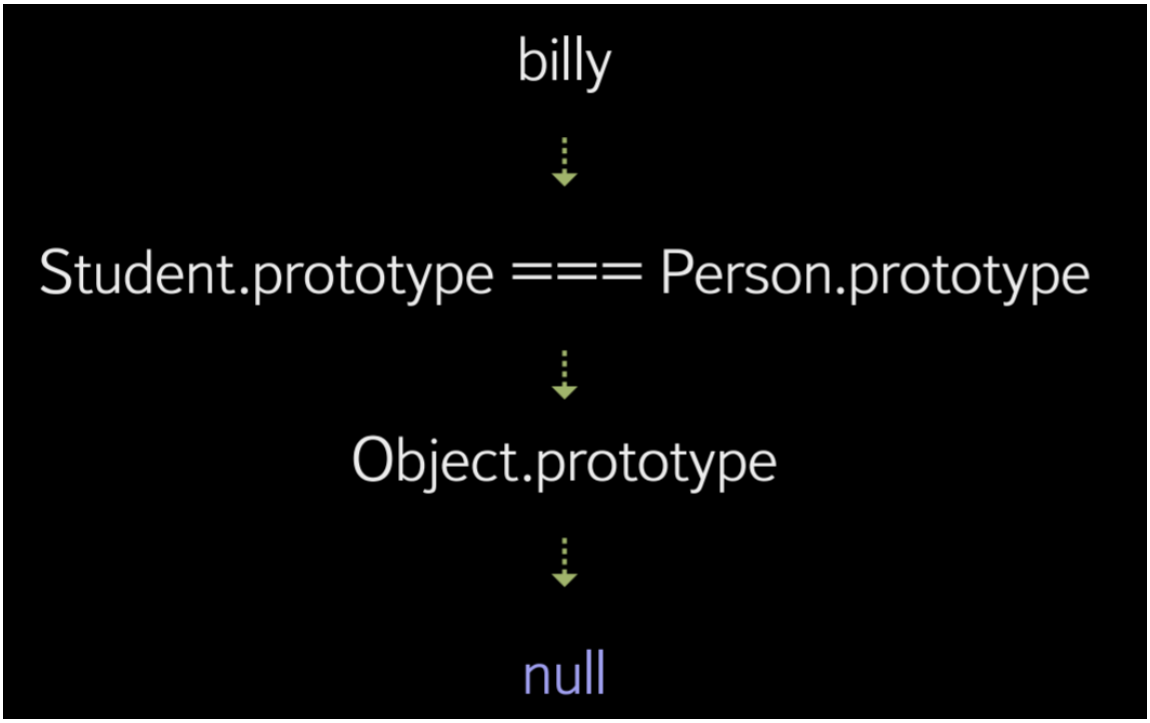
У нас есть некоторый конструктор студентов, который принимает на вход имя студента и записывает его в поле name. А также хранилище, в поле .prototype нашего конструктора мы поместили объект, который хотим видеть в качестве прототипа для всех вновь создаваемых студентов. Там сейчас один метод – sleep.
Допустим, у нас есть более абстрактный конструктор, который также имеет хранилище и в котором расположен другой метод – getName.
В данном случае нам бы не хотелось дублировать реализацию этого метода в хранилище конструктора студентов. Для этого мы можем пойти самым простым путем и хранилище конструктора студентов сделать на основе хранилища конструктора Person: присваиваем в специальное поле prototype ссылку на хранилище конструктора Person. Далее, мы можем расширить наше хранилище для студентов более специфичным для них методом – методом sleep. Итак, мы можем создавать теперь новых студентов, которые будут иметь доступ к методу sleep из своего хранилища и к методу getName, которое изна-

```
function Student(name) {
this.name = name;
}
Student.prototype = Person.prototype;
Student.prototype.sleep = function () {};
```

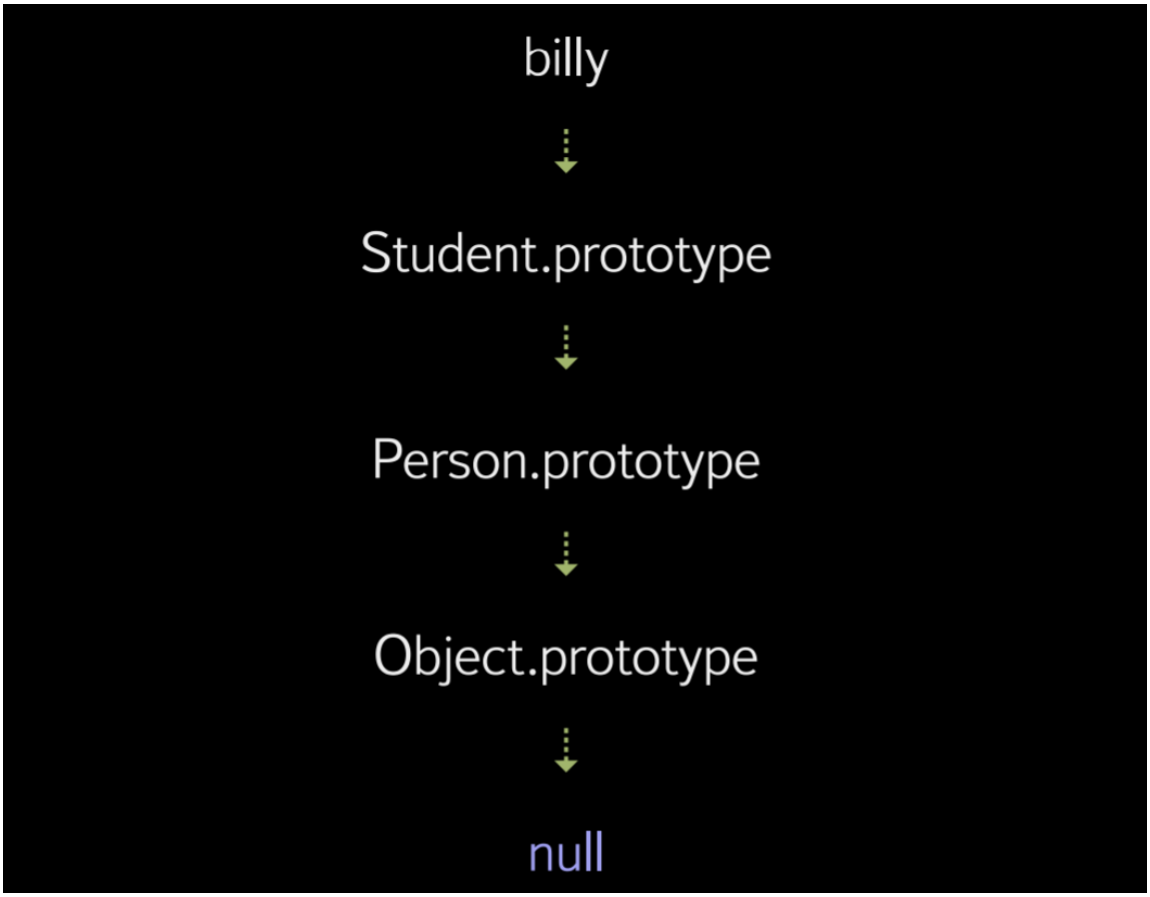
чально было в хранилище конструктора `Person`.
Данный способ имеет подводный камень. Если мы в своей программе попробуем использовать объекты другого типа, создадим для них конструктор и в хранилище этого конструктора запишем ссылку на то же самое хранилище, которое связано с конструктором `Person`, безусловно, наш новый объект – преподаватель – получит доступ к методам из этого хранилища `Person`. Но мы с удивлением обнаружим, что преподаватель получит доступ и к методам, которые мы ранее определили только для студентов, например к методу `sleep`.

```
function Student(name) {
    this.name = name;
}
Student.prototype = Person.prototype;
Student.prototype.sleep = function () {};

function Lecturer(name) {
    this.name = name;
}
Lecturer.prototype = Person.prototype;

var sergey = new Lecturer('Sergey');

sergey.sleep(); // zzzZZ ...
```

Это происходит потому, что все три этих хранилища хранят сейчас ссылки на один и тот же объект. Наша цепочка прототипов выглядит примерно так.



Это нежелательное поведение и нам бы хотелось, чтобы цепочка выглядела примерно так.



1.3.1. Метод `create`

Решить эту проблему нам поможет специальный метод `create`. Вместо обычного присваивания мы в хранилище для конструктора студентов будем записывать объект, который нам этот метод возвращает. Такой же трюк мы можем проделать и с конструктором преподавателя. Попробуем теперь создать нового преподавателя и вызвать у него метод, который мы определили только для студентов. Благодаря методу `create`, мы увидим, что преподаватели не будут иметь доступ к методам, которые мы определили только для студентов.

```
function Student(name) {
    this.name = name;
}

Student.prototype = Object.create(Person.prototype);
Student.prototype.sleep = function () {};


function Lecturer(name) {
    this.name = name;
}
Lecturer.prototype = Object.create(Person.prototype);

var sergey = new Lecturer('Sergey');
sergey.sleep(); //TypeError: sergey.sleep is not a function
```

Попробуем разобраться, как работает метод `create`. Метод создает пустой объект, прототипом которого становится объект, переданный первым аргументом. Есть прототип для фруктов, которые хранит единственное поле. Оно говорит нам о том, что все фрукты полезные. И на основе этого прототипа фруктов мы будем создавать новые фрукты при помощи метода `create`, передавая в качестве аргумента наш прототип. Так мы можем создать яблоко и проверить, что оно действительно полезное, несмотря на то, что этого поля у самого яблока нет. Оно есть у его прототипа.

```
var fruitProto = {
isUsefull: true
}
var apple = Object.create(fruitProto);
apple.isUsefull; // true
```

Внутри метод `create` устроен достаточно просто. Он создает простейшие конструкторы из возможных, а именно: пустую функцию. Далее, в хранилище этого конструктора он записывает ссылку на тот объект, который мы передаем в качестве первого аргумента, тот объект, который хотим видеть в качестве прототипа для всех создаваемых объектов. Далее, он при помощи этого конструктора создает новый объект и возвращает его. Таким образом, все вновь

создаваемые объекты будут иметь в качестве прототипа тот объект, который мы передаем первым аргументом.

```
var apple = Object.create(fruitProto);

Object.create = function(prototype) {
    // Простейший конструктор пустых объектов
    function EmptyFunction() {};
    EmptyFunction.prototype = prototype;
    return new EmptyFunction();
};
```

В метод `create` мы можем передавать не только объекты, но и, например, значение `null`. В этом случае мы создадим объект, в качестве прототипа которого не будет выступать ни один из объектов, даже глобальный прототип для всех объектов. И мы не получим доступ к методам из этого глобального прототипа. Метод `create` помогает нам связать два хранилища разных конструкторов так, чтобы они не ссылались на один и тот же объект. И хранилище для конструктора студентов будет представлять из себя отдельный объект, но во внутреннем поле `prototype` которого лежит ссылка на другое хранилище – хранилище конструктора `Person`. Далее мы можем расширить хранилище для студентов специфичными для них методами.

```
function Student(name) {
    this.name = name;
}
Student.prototype = Object.create(Person.prototype);
Student.prototype.sleep = function () {};
```

И здесь мы допустили ту же самую ошибку, что и ранее. Мы полностью перезаписали хранилище студентов и забыли о поле конструктора.
Давайте его вернем. Достаточно просто присвоить в него ссылку на функцию.

```
function Student(name) {
    this.name = name;
}
Student.prototype = Object.create(Person.prototype);
Student.prototype.sleep = function () {};

Student.prototype.constructor = Student;
```

Итоговое решение нашей задачи выглядит примерно так.

```
function Person() {
    this.type = 'human';
}

Person.prototype.getName = function () {
    return this.name;
};

function Student(name) {
    this.name = name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.sleep = function () {};

Student.prototype.constructor = Student;

var billy = new Student('Billy');
```

1.4. Инспектирование связей между объектами, конструкторами и прототипами

1.4.1. `getPrototypeOf`

Первый способ — это метод `getPrototypeOf`. На вход он принимает в себя объект, а на выходе дает ссылку на прототип для этого объекта. В данном случае

у нас есть объект студента, в качестве прототипа для которого выступает объект `person`. Вызвав метод `getPrototypeOf` и передав туда ссылку на объект студента, мы на выходе получим на объект `person`.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
}

var person = {
    type: 'human',
    getName: function () {}
}

Object.getPrototypeOf(student) === person; // true
```

1.4.2. `isPrototypeOf`

Следующий способ инспектирования связей между объектами и прототипом предлагает нам метод `isPrototypeOf`. Допустим, у нас есть некий конструктор студентов, который на вход принимает имя студента и сохраняет его в поле `name` для каждого студента; хотим к каждому студенту привязать некоторый прототип. Мы помещаем этот прототип в поле `prototype`, в хранилище, и делаем его объектом на основе другого хранилища — конструктора `person` при помощи метода `create`. Далее, мы в наше хранилище добавляем специфичный для студентов метод `sleep` и восстанавливаем конструктор. Попробуем теперь создать нашего студента и воспользоваться для проверки связи между созданным студентом и его прототипом методом is`prototype`Of. Данный метод отвечает на вопрос: «Является ли объект прототипом для того объекта, который мы передаем в качестве аргументов?»

```
function Student(name) {
    this.name = name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.sleep = function () {};

Student.prototype.constructor = Student;

var billy = new Student('Billy');

Student.prototype.isPrototypeOf(billy); // true
```

Более того, данный метод позволяет инспектировать не только прямую связь, но и связь конечного объекта с одним из прототипов цепочки. Таким образом, он даст утвердительный ответ и на вопросы: «Является ли объект, который лежит в хранилище конструктора `person`, прототипом для `Billy`?» и «Является ли глобальный прототип для всех объектов прототипом для `Billy`?»

```
Person.prototype.isPrototypeOf(billy); // true

Object.prototype.isPrototypeOf(billy); // true
```

1.4.3. `instanceof`

Следующий способ инспектирования связей между объектами и конструкторами — специальный оператор `instanceof`. Он позволяет ответить на вопрос: «Является ли объект объектом определенного конструктора?» В данном случае мы создали нового студента `Billy` на основе конструктора Student и проверяем, является ли `Billy` объектом этого конструктора.

```
function Student(name) {
    this.name = name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.sleep = function () {};

Student.prototype.constructor = Student;

var billy = new Student('Billy');

billy instanceof Student; // true
```

Но этот оператор работает несколько сложнее, и он ответит, что `Billy` является и объектом конструктора `Person`, хотя напрямую это не так.

```
billy instanceof Person; // true

billy instanceof Object; // true
```

Но можно заметить, что хранилище, связанное с конструктором `Person`, лежит в цепочке прототипов до `Billy`. Если мы немножечко переформулируем то, как работает этот оператор, все встанет на свои места. Можно сказать, что он отвечает на вопрос: «Является ли `Billy` студентом?» или: «Является ли `Billy` личностью?» Более того, `Billy`, конечно же, является объектом — тоже правда.

Давайте разберем, как работает оператор `instanceof`. Например, как он проверяет, связаны ли между собой объект `Billy` с конструктором `Person`. Вначале он проверяет, является ли прототипом для `Billy` объект, который хранится в хранилище этого конструктора. И это не так. Тогда он проверяет следующую гипотезу: «А, может быть, прототипа у `Billy` совсем нет, и там хранится значение `null`?» Эта гипотеза также не подтверждается, и тогда он идет по цепочке прототипов. Он проверяет, является ли прототип прототипа `Billy` объектом, который хранится в хранилище конструктора `Person`. На этот раз гипотеза подтверждается, и в этом случае данный оператор отвечает нам `true`.

```
billy instanceof Person;
billy.__proto__ === Person.prototype;
// false -> Может, там null?
billy.__proto__ === null;
// false -> Идём дальше по цепочке
billy.__proto__.__proto__ === Person.prototype;
// true -> Возвращаем true
```

Если мы попробуем проинспектировать объект с самой короткой цепочкой прототипов, оператор `instanceof` возвращает нам `false`. Мы создаем объект и проверяем, что этот объект — это действительно объект, но оператор `instanceof` возвращает нам `false`. Если мы еще раз посмотрим, как работает этот оператор, все встанет на свои места. Итак, в первую очередь, он проверяет, является ли прототипом для нашего одинокого объекта глобальный прототип для всех объектов. Это не так. Тогда он проверяет: «Возможно, у нашего одинокого объекта совсем нет прототипа, и во внутреннем поле `prototype` лежит значение `null`?» В этом случае оператор `instanceof` при положительном подтверждении такой гипотезы возвращает нам `false`.

```
var foreverAlone = Object.create(null);

foreverAlone instanceof Object; // false

Object.create(null).__proto__ === Object.prototype;
// false -> Может, там null?

Object.create(null).__proto__ === null;
// true -> Так и есть, возвращаем false!
```

1.5. Решение проблемы дублирования кода в конструкторах

У нас есть конструктор студентов, конструктор преподавателей и конструктор личностей. Если мы посмотрим на конструкторы преподавателей и студентов, мы увидим, что они похожи и выполняют одинаковую работу: принимают в

качестве аргумента имя студента или преподавателя и сохраняют его в поле `name`.

```
function Student(name) {
    this.name = name;
}

function Lecturer(name) {
    this.name = name;
}

function Person() {
    this.type = 'human';
}
```

Нам бы хотелось избежать этого дублирования. Проще всего — вынести этот общий код в отдельный конструктор `Person`. Перенесем туда строчку, которая сохраняет имя студента или преподавателя в конструктор `Person`.

```
function Student(name) {
    this.name = name;
}

function Lecturer(name) {
    this.name = name;
}

function Person() {
    this.type = 'human';
}
```

Если мы сейчас попытаемся создать нового студента и передадим туда имя, мы не получим желаемого результата, так как мы забрали этот код из конструктора для студентов и переместили его в конструктор для личностей. Поэтому нам необходимо немножечко изменить реализацию конструктора студентов и добавить туда вызов конструктора личностей. Мы можем это сделать при помощи метода `call`. Вызываем наш конструктор личностей при помощи этого

метода и в качестве контекста передаем туда `this`, а в качестве второго аргумента передаем имя создаваемого нами студента.

```
function Person(name) {
    this.type = 'human';
    this.name = name;
}

function Student(name) {
    // this ссылается на новый объект студента
    Person.call(this, name);
}

var billy = new Student('Billy');

console.info(billy.name); // Billy
```

Благодаря тому, что в конструкторе `this` будет ссылаться на новый создаваемый объект, в нашем случае — на нового студента, мы вызовем конструктор `Person` с контекстом в виде этого студента, и таким образом передаваемое имя мы положим в поле name именно для студента. Все будет работать.

1.6. Вызов затеняемого метода в затеняющем

Допустим, у нас есть некоторый конструктор `Person`, конструктор личностей, который на вход принимает имя этих личностей и кладет в поле name. Также мы определяем прототип для всех личностей с методом `getName`, который возвращает это имя. Положим этот прототип в хранилище конструктора `Person`.

```
function Person(name) {
    this.name = name;
}

Person.prototype.getName = function () {
    return this.name;
}
```

Далее мы захотим использовать в нашей программе объекты другого типа и создадим конструктор для них. Прототипом для них мы сделаем объект на основе прототипа для личностей, но захотим изменить метод `getName`. Нам не хочется дублировать тот код, который есть в методе `getName` прототипа личностей, а лишь дополнить и добавлять к нему некую строку.
Вначале нам может показаться, что мы можем вызвать просто метод `getName` из прототипа `Person` внутри метода `getName` прототипа для студентов. Но в этом случае произойдет рекурсивный вызов того же самого метода. И интерпретатор нам скажет, что количество вызовов заполнило весь стек.

```
1  Student.prototype = Object.create(Person.prototype);
2
3  Student.prototype.getName = function () {
4      return 'Student ' + this.getName();
5  };
6
7  var billy = new Student('Billy');
8
9  billy.getName(); // RangeError: Maximum call stack size exceeded
```

Когда мы вызываем метод `getName` у созданного нами студента, например, `Billy`, внутри этого метода `this` будет ссылаться, собственно, на этого самого студента. Так как метода `getName` у самого `Billy` нет, он пойдет искать его в прототипе и найдет метод в прототипе, который хранится в хранилище конструктора `Student` в поле `Student.prototype`. Фактически метод будет вызывать сам себя.
Каким образом мы можем решить эту проблему? Самое простое — использовать другое название метода вместо эффекта затенения. В этом случае все будет работать. Мы будем вызывать метод с другим названием, например, getStudentName, заходить внутрь этого метода, `this` внутри него будет попрежнему ссылаться на объект `Student`, то есть на `Billy`, попробуем найти у `Billy` метод `getName`, не найдем его там, пройдем по цепочке прототипов в объект, который хранится в хранилище конструктора студентов Student.`prototype`. И там мы этого метода не найдем, проследуем дальше по цепочке прототипов и уже перейдем в хранилище, которое хранится в конструкторе `Person`. Там мы этот метод находим, спокойно его вызываем, возвращаем имя студента, добавляем к нему наш необходимый префикс, и все работает, как нужно.

```
function Person(name) {
    this.name = name;
}

Person.prototype.getName = function () {
    return this.name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.getStudentName = function () {
    return 'Student ' + this.getName();
};

var billy = new Student('Billy');

billy.getStudentName();
```

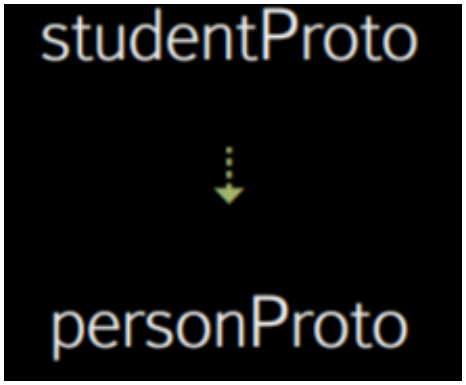
Более элегантным способом будет использование метода `call`. Мы можем напрямую в затеняющем методе вызывать затеняемый при помощи этого метода `call`, но передавать туда текущий контекст. А текущий контекст будет ссылаться на создаваемого объекта конструктора студентов, а именно на студента. Таким образом мы вызываем затеняемый метод от лица этого самого студента, получаем имя этого студента и добавляем к нему префикс `student`.

```
function Person(name) {
    this.type = 'human';
    this.name = name;
}

Person.prototype.getName = function () {
    return this.name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.getName = function () {
    return 'Student ' + Person.prototype.getName.call(this);
};
```

1.7. Сравнение трёх подходов к конструированию объектов: функции-конструкторы, метод `create`, «Классы»

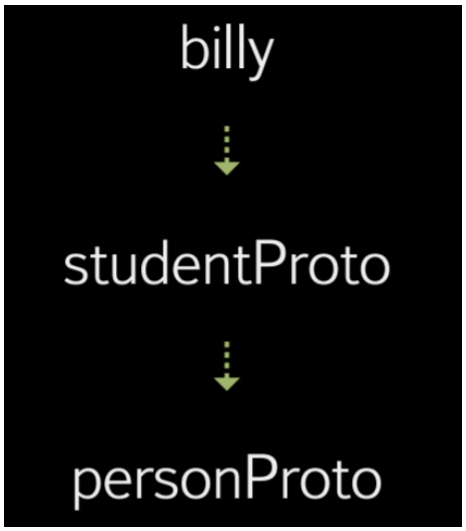
Для конструирования объектов и построения связей между ними достаточно лишь использовать только метод `create`.
Давайте снова попробуем создать конструктор для студентов. Для этого нам вновь понадобится ряд прототипов, выстроенных в цепочку. На этот раз мы воспользуемся только методом `create` и обычными объектами, так как любой прототип по факту — это обычный объект. Итак, для начала нам понадобится прототип личности. Создадим простой объект `personProto` и добавим туда метод `getName`, который будет возвращать имя нашей личности. На основе этого прототипа создадим более специфичный прототип уже для студентов. При помощи метода `create` мы свяжем два этих прототипа в цепочку. Далее мы можем расширить наш прототип для студентов уже специфичными для студентов методами, например, методом `sleep`.

```
var personProto = {
    getName: function () {
        return this.name;
    }
};
var studentProto =
 ↪  Object.create(personProto);

studentProto.sleep = function () {};
```



Далее на основе прототипа для студентов мы уже будем создавать студентов. Для этого снова воспользуемся методом `create`. Таким образом, благодаря этому методу мы встроим нашего студента в цепочку прототипов. И далее всё, что нам останется, это только дополнить нашего студента полями, которые необходимы каждому конкретному студенту с конкретными значениями. В данном случае мы присваиваем нашему студенту имя `Billy`.

```
var billy =
 ↪  Object.create(studentProto);

billy.name = 'Billy';
```



Благодаря такому подходу нам понадобилось значительно меньше строчек кода, чем в классическом, чтобы научить нашу программу создавать новые объекты студентов. Но и здесь мы можем произвести улучшение. Для этого нам понадобится ещё одна возможность метода `create`, а именно — данный метод принимает не один аргумент, а два. В качестве второго аргумента вы можете

передать необходимые поля с их начальными значениями и характеристиками, которые вы хотите видеть при создании объекта в итоговом объекте.

```
var apple = Object.create(fruit, {
    shape: { value: 'round', writable: false },
    color: { value: 'Green' },
    amount: { writable: true }
});

apple.amount = 'half';
```

Посмотрим, каким образом мы можем использовать данную возможность. Чтобы создавать наших студентов в одну строку, нам понадобится дополнительная функция-помощник, назовём её `create` и положим в наш прототип для студентов. Иногда такие функции называют фабриками — фабриками объектов. Данная функция будет на вход принимать в себя имя студента, а внутри себя вызывать метод `Object.create`, в качестве контекста передавать `this` и в качестве второго аргумента передавать набор полей, которые мы хотим видеть у студента, а именно: мы хотим у него видеть поле `name`, и мы сразу заполняем его тем, что нам приходит в фабрику. Таким образом мы можем создавать новых студентов в одну строчку, как и при классическом подходе при помощи оператора `new`. В данном случае мы просто вызываем наш метод `create`, который лежит в прототипе студента, передаём туда имя и получаем новый объект, нового студента с этим именем.

```
var personProto = {};

personProto.getName = function () { return this.name; }

var studentProto = Object.create(personProto);

studentProto.sleep = function () {};

studentProto.create = function (name) {
    return Object.create(this, {
        name: { value: name }
    });
}

var billy = studentProto.create('Billy');
```

Единого мнения, какой из этих подходов лучше, у разработчиков нет.
Также есть новая версия спецификаций, которая вводит новую синтаксическую конструкцию, а именно «классы». И вместо функций-конструкторов мы можем определить класс и на основе этого класса создавать новые объекты.

```
function Student(name) {
    this.name = name;
}

Student.prototype.getName = function () {
    return this.name
};

class Student {
    constructor(name) {
        this.name = name;
    }

    getName() {
        return this.name;
    }
}
```

Классы - это не более, чем обычные функции-конструкторы. У них также есть специальное поле `prototype` — хранилище для прототипов, для всех вновь создаваемых этим классом объектов. Также если мы посмотрим тип классов, то мы увидим, что это обычная функция.

```
class Student {
// ...
}

var billy = new Student('Billy');

billy.getName(); // Billy

Student.prototype.isPrototypeOf(billy); // true

typeof Student; // function
```

Таким образом, дискуссия трансформировалась в другую: что выбрать, классы или метод `Object.create`? Наиболее полно её раскрыл в статье разработчик Eric Elliott, и я рекомендую вам всем ознакомиться с ней.

Конструкторы

Курс: JavaScript, часть 2: прототипы и асинхронность

20 февраля 2018 г.

Оглавление

Глава 1

Конструкторы

1.1. Конструкторы

Обычно в программе мы работаем не с одним конкретным объектом, а с целой коллекцией однотипных объектов. И нам необходимо уметь создавать объекты того же типа. Создание нового объекта такого же типа — достаточно громоздкая операция.
Можно вынести процесс создания новых объектов в конструктор.
Пусть это будет обычная функция — назовем ее `createStudent`, которая на вход принимает в качестве параметра имя студента, а на выходе отдает нам новый объект с заполненными полями и необходимыми методами.

```
1  function createStudent(name) {
2      return {
3          name: name,
4          sleep: function () {
5              console.info('zzzZZ ...');
6          }
7      };
8  }
9  var billy = createStudent('Billy');
10 var willy = createStudent('Willy');
```

Данное решение простое, но каждый раз при вызове конструктора мы будем создавать новую функцию, которая будет реализовывать метод `sleep`. Можно вынести его в прототип. Для этого создадим новый объект `studentProto`,

который будет являться прототипом для всех вновь создаваемых студентов, и перенесем туда наш метод `sleep`. После этого нам необходимо добавить туда вызов метода `setPrototypeOf`, который будет привязывать новых студентов к уже созданному нами прототипу. Каждый студент будет иметь доступ к методам, которые хранятся в прототипе для него.

```
var studentProto = {
    sleep: function () {
        console.info('zzzZZ ...');
    }
};
function createStudent(name) {
    var student = {
        name: name
    };
    Object.setPrototypeOf(student, studentProto);
    return student;
}
```

Мы можем воспользоваться уже готовым механизмом для создания конструкторов. Любая функция может быть конструктором, если мы вызовем ее при помощи специального оператора `new`. Функция `createStudent` может стать конструктором сама, но нам необходимо переписать ее реализацию и оставить только передаваемый аргумент, который будет хранить имя вновь создаваемого студента; всю остальную работу за нас сделает интерпретатор. Перед тем как исполнить код нашего конструктора, он создаст новый объект и присвоит его в переменную `this`. Далее мы заполним этот объект полями, и в конце интерпретатор за нас вернет объект, хранящийся по ссылке `this`. Можно сказать, что при вызове функции как конструктора с оператором `new` `this` внутри этой функции при исполнении будет указывать на вновь создаваемый объект.

```
var billy = new createStudent('Billy');

function createStudent(name) {
    // var this = {};
    this.name = name;
    // return this;
}
```

Такой код уже будет работать, но читается он не очень хорошо; переименуем нашу функцию просто в функцию `Student`.

```
function Student(name) {
    this.name = name;
}
var billy = new Student('Billy');
```

Функции-конструкторы принято именовать с заглавной буквы. Почему это важно?
Что произойдет, если мы попытаемся вызвать случайно, например, нашу функцию-конструктор как обычную функцию без оператора `new`? В этом случае переданное значение имени студента не будет записано в новый объект студента, а будет записано в глобальный объект в поле `name`, так как, вызывая функцию, `this` по умолчанию будет ссылаться на глобальный объект. Мы можем следовать соглашению и дополнительно включить строгий режим интерпретации, который защитит нас от такого поведения. В этом случае `this` будет иметь значение `undefined`, и мы не сможем присвоить в него никакие поля.

```
function Student(name) {
    this.name = name;
}
var billy = Student('Billy'); // Поле появится в глобальном объекте!

window.name === 'Billy'; // true

'use strict'; // TypeError: Cannot set property 'name' of undefined
```

Давайте попробуем вмешаться в работу конструктора, в работу интерпретатора и как-то поменять поведение конструктора. Например, мы захотим сами возвращать какой-то сконструированный объект. Можем ли мы это сделать? В данном случае интерпретатор нам полностью доверяет и вернет тот объект, который мы возвращаем при помощи оператора `return`.

```
function Student(name) {
    this.name = name;
    return {
        name: 'Muahahahahaha!'
    };
}
var billy = new Student('Billy');
console.info(billy.name); // Muahahahahaha
```

Но если мы попытаемся вернуть из конструктора какое-то примитивное значение — число, строку или `null`, в этом случае интерпретатор просто проигнорирует эту строку и будет работать как раньше: он будет возвращать вновь создаваемый объект.

```
function Student(name) {
this.name = name;
return null; // Evil mode on!
}
var billy = new Student('Billy');
console.info(billy.name);
// Billy
```

1.2. Конструкторы и прототипы

Допустим, у нас есть конструктор студентов и нам хотелось бы добавить в него несколько методов. Логично было бы хранить их в прототипе. Для того чтобы автоматически привязывать этот прототип для всех вновь создаваемых студентов, нам необходимо поместить его в хранилище. Оно есть у каждой функции конструктора в специальном поле prototype. Конструктор в момент исполнения выполняет дополнительный шаг: привязывает тот объект, который мы поместили в хранилище, в качестве прототипа для всех вновь создаваемых объектов; создавая новых студентов при помощи нашего конструктора, мы увидим, что у каждого из них внутреннее поле `prototype` будет ссылаться на тот объект, который мы ранее поместили в хранилище.

```
Student.prototype = {
    sleep: function () {}
};
function Student(name) {
    // var this = {};
    this.name = name;
    // Object.setPrototypeOf(this, Student.prototype);
    // return this;
}
```

Данное хранилище может вам напомнить другое хранилище, а именно то, которое расположено в специальном поле `prototype` функции `Object`. И более того, автоматически каждый создаваемый объект в JavaScript имеет в качестве прототипа объект этого хранилища. Изначально нам было не очень очевидно, каким же образом осуществлялась данная привязка, ведь мы создавали объекты при помощи литеральной конструкции, а не конструктора и оператора `new`. Но под капотом интерпретатор вызывает тот же самый конструктор `Object` оператором `new`.
Давайте подробнее поговорим про специальное поле `.prototype`:

- есть у каждой функции;
- хранит объект;
- имеет смысл только при вызове функции как конструктора;
- имеет вложенное поле `.constructor` (неперечисляемое, хранит ссылку на саму функцию).

Обращаясь к полю `.constructor`, мы можем получить доступ к конструктору, т.е можем, например, выяснить имя конструктора конкретного объекта. Например, если мы сконструировали нового студента `Billy` на основе конструктора Student, мы можем обратиться к этому полю у `Billy`. У `Billy` этого поля нет, но оно есть в прототипе, который, как мы знаем, изначально хранится в хранилище Student.`prototype`. И так как данное поле хранит ссылку на функцию, мы можем посмотреть имя этой функции, обратившись к полю name.

```
function Student(name) {
    this.name = name;
}

Student.prototype.constructor === Student; // true
var billy = new Student('Billy');
console.info(billy.constructor.name); // Student
```

Важно помнить о поле `.constructor`, так как мы очень легко можем его перезаписать. В нашем случае мы это и сделали. Мы просто поместили в поле .prototype, в это хранилище, новый объект, тем самым уничтожив поле `.constructor`.

```
function Student(name) {
    this.name = name;
}
Student.prototype = {
    sleep: function () {}
};
```

Чтобы этого не произошло, достаточно не перезаписывать тот объект, который хранится изначально в этом хранилище, а дополнять его новыми методами. Таким образом, мы получим на основе этого конструктора новые объекты. Они будут иметь доступ как к методам из прототипа, так и к специальному полю `.constructor`.

```
function Student(name) {
    this.name = name;
}

Student.prototype.sleep = function () {
    console.info('zzzZZ ...');
}

var Billy = new Student('Billy');
billy.sleep(); // zzzZZ ...
billy.constructor === Student; // true
```

```
function Student(name) {
    this.name = name;
}
Student.prototype.sleep = function () {};
```

```
function Person() {
    this.type = 'human';
}
Person.prototype.getName = function () {
    return this.name;
}
```

1.3. Конструкторы и цепочки прототипов

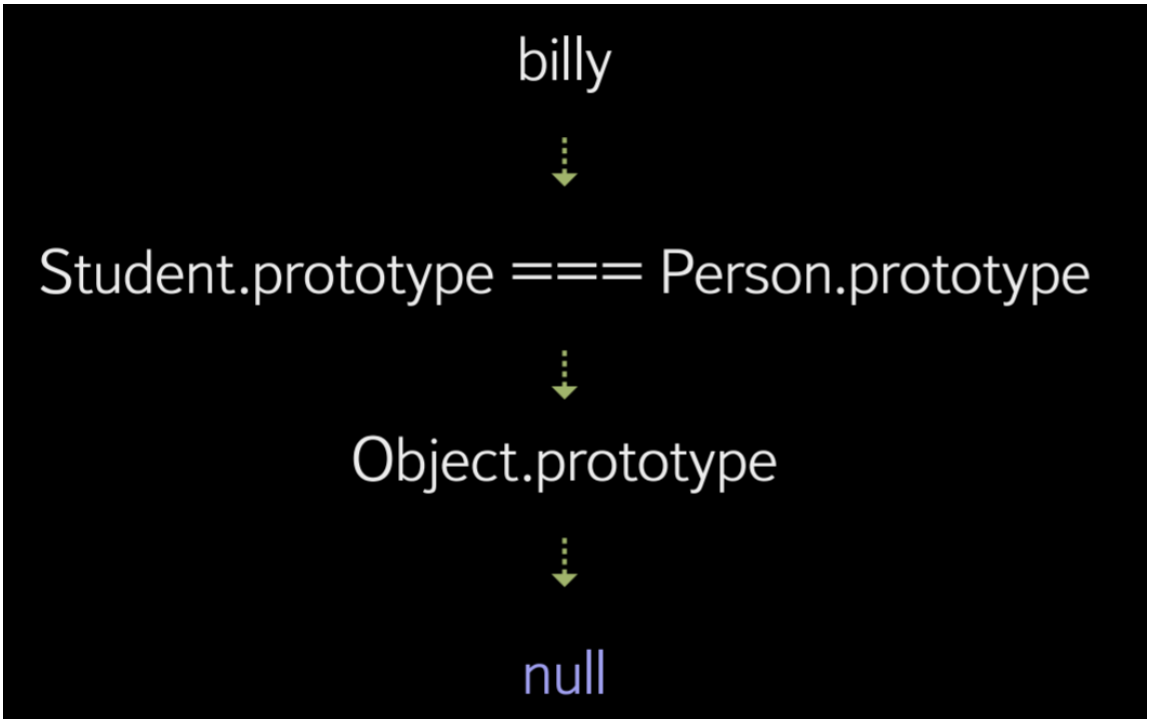
У нас есть некоторый конструктор студентов, который принимает на вход имя студента и записывает его в поле name. А также хранилище, в поле .prototype нашего конструктора мы поместили объект, который хотим видеть в качестве прототипа для всех вновь создаваемых студентов. Там сейчас один метод – sleep.
Допустим, у нас есть более абстрактный конструктор, который также имеет хранилище и в котором расположен другой метод – getName.
В данном случае нам бы не хотелось дублировать реализацию этого метода в хранилище конструктора студентов. Для этого мы можем пойти самым простым путем и хранилище конструктора студентов сделать на основе хранилища конструктора Person: присваиваем в специальное поле prototype ссылку на хранилище конструктора Person. Далее, мы можем расширить наше хранилище для студентов более специфичным для них методом – методом sleep. Итак, мы можем создавать теперь новых студентов, которые будут иметь доступ к методу sleep из своего хранилища и к методу getName, которое изна-

```
function Student(name) {
this.name = name;
}
Student.prototype = Person.prototype;
Student.prototype.sleep = function () {};
```

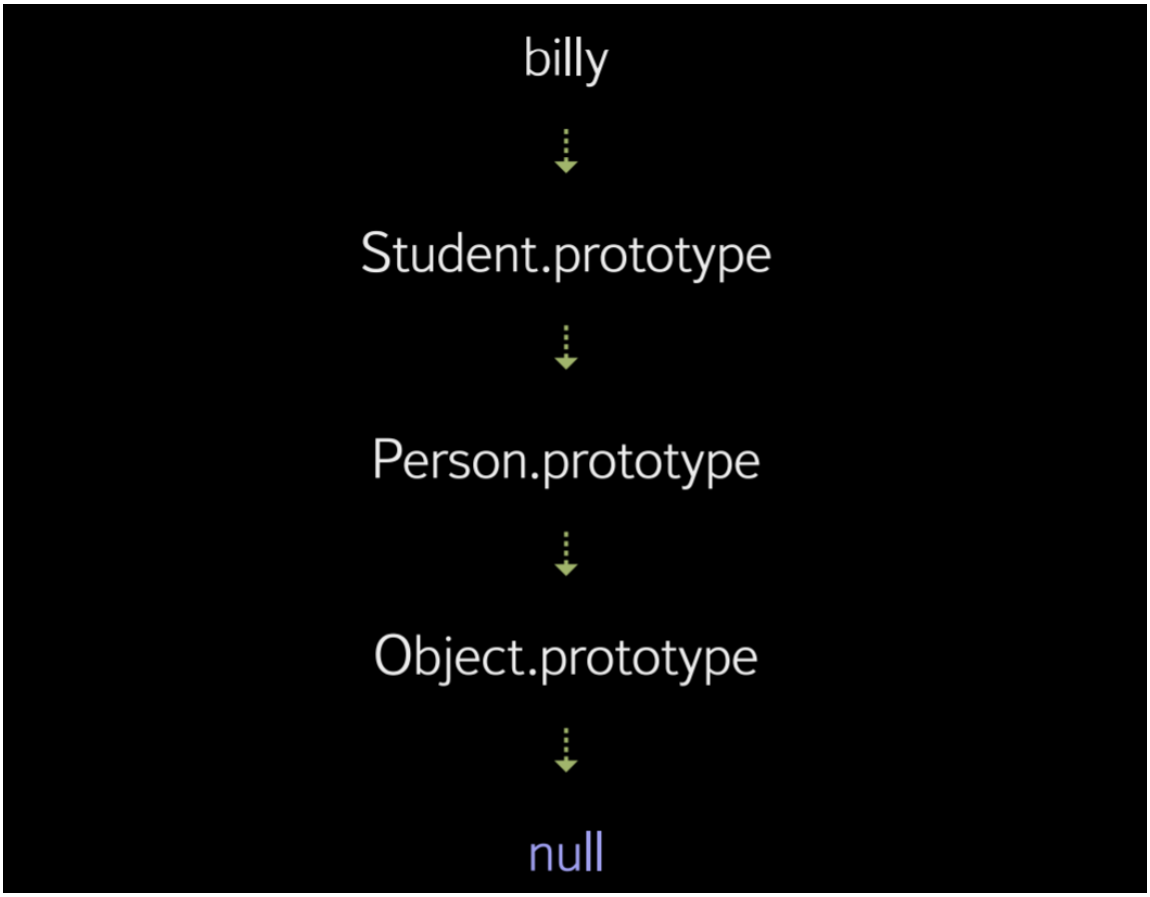
чально было в хранилище конструктора `Person`.
Данный способ имеет подводный камень. Если мы в своей программе попробуем использовать объекты другого типа, создадим для них конструктор и в хранилище этого конструктора запишем ссылку на то же самое хранилище, которое связано с конструктором `Person`, безусловно, наш новый объект – преподаватель – получит доступ к методам из этого хранилища `Person`. Но мы с удивлением обнаружим, что преподаватель получит доступ и к методам, которые мы ранее определили только для студентов, например к методу `sleep`.

```
function Student(name) {
    this.name = name;
}
Student.prototype = Person.prototype;
Student.prototype.sleep = function () {};

function Lecturer(name) {
    this.name = name;
}
Lecturer.prototype = Person.prototype;

var sergey = new Lecturer('Sergey');

sergey.sleep(); // zzzZZ ...
```

Это происходит потому, что все три этих хранилища хранят сейчас ссылки на один и тот же объект. Наша цепочка прототипов выглядит примерно так.



Это нежелательное поведение и нам бы хотелось, чтобы цепочка выглядела примерно так.



1.3.1. Метод `create`

Решить эту проблему нам поможет специальный метод `create`. Вместо обычного присваивания мы в хранилище для конструктора студентов будем записывать объект, который нам этот метод возвращает. Такой же трюк мы можем проделать и с конструктором преподавателя. Попробуем теперь создать нового преподавателя и вызвать у него метод, который мы определили только для студентов. Благодаря методу `create`, мы увидим, что преподаватели не будут иметь доступ к методам, которые мы определили только для студентов.

```
function Student(name) {
    this.name = name;
}

Student.prototype = Object.create(Person.prototype);
Student.prototype.sleep = function () {};


function Lecturer(name) {
    this.name = name;
}
Lecturer.prototype = Object.create(Person.prototype);

var sergey = new Lecturer('Sergey');
sergey.sleep(); //TypeError: sergey.sleep is not a function
```

Попробуем разобраться, как работает метод `create`. Метод создает пустой объект, прототипом которого становится объект, переданный первым аргументом. Есть прототип для фруктов, которые хранит единственное поле. Оно говорит нам о том, что все фрукты полезные. И на основе этого прототипа фруктов мы будем создавать новые фрукты при помощи метода `create`, передавая в качестве аргумента наш прототип. Так мы можем создать яблоко и проверить, что оно действительно полезное, несмотря на то, что этого поля у самого яблока нет. Оно есть у его прототипа.

```
var fruitProto = {
isUsefull: true
}
var apple = Object.create(fruitProto);
apple.isUsefull; // true
```

Внутри метод `create` устроен достаточно просто. Он создает простейшие конструкторы из возможных, а именно: пустую функцию. Далее, в хранилище этого конструктора он записывает ссылку на тот объект, который мы передаем в качестве первого аргумента, тот объект, который хотим видеть в качестве прототипа для всех создаваемых объектов. Далее, он при помощи этого конструктора создает новый объект и возвращает его. Таким образом, все вновь

создаваемые объекты будут иметь в качестве прототипа тот объект, который мы передаем первым аргументом.

```
var apple = Object.create(fruitProto);

Object.create = function(prototype) {
    // Простейший конструктор пустых объектов
    function EmptyFunction() {};
    EmptyFunction.prototype = prototype;
    return new EmptyFunction();
};
```

В метод `create` мы можем передавать не только объекты, но и, например, значение `null`. В этом случае мы создадим объект, в качестве прототипа которого не будет выступать ни один из объектов, даже глобальный прототип для всех объектов. И мы не получим доступ к методам из этого глобального прототипа. Метод `create` помогает нам связать два хранилища разных конструкторов так, чтобы они не ссылались на один и тот же объект. И хранилище для конструктора студентов будет представлять из себя отдельный объект, но во внутреннем поле `prototype` которого лежит ссылка на другое хранилище – хранилище конструктора `Person`. Далее мы можем расширить хранилище для студентов специфичными для них методами.

```
function Student(name) {
    this.name = name;
}
Student.prototype = Object.create(Person.prototype);
Student.prototype.sleep = function () {};
```

И здесь мы допустили ту же самую ошибку, что и ранее. Мы полностью перезаписали хранилище студентов и забыли о поле конструктора.
Давайте его вернем. Достаточно просто присвоить в него ссылку на функцию.

```
function Student(name) {
    this.name = name;
}
Student.prototype = Object.create(Person.prototype);
Student.prototype.sleep = function () {};

Student.prototype.constructor = Student;
```

Итоговое решение нашей задачи выглядит примерно так.

```
function Person() {
    this.type = 'human';
}

Person.prototype.getName = function () {
    return this.name;
};

function Student(name) {
    this.name = name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.sleep = function () {};

Student.prototype.constructor = Student;

var billy = new Student('Billy');
```

1.4. Инспектирование связей между объектами, конструкторами и прототипами

1.4.1. `getPrototypeOf`

Первый способ — это метод `getPrototypeOf`. На вход он принимает в себя объект, а на выходе дает ссылку на прототип для этого объекта. В данном случае

у нас есть объект студента, в качестве прототипа для которого выступает объект `person`. Вызвав метод `getPrototypeOf` и передав туда ссылку на объект студента, мы на выходе получим на объект `person`.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
}

var person = {
    type: 'human',
    getName: function () {}
}

Object.getPrototypeOf(student) === person; // true
```

1.4.2. `isPrototypeOf`

Следующий способ инспектирования связей между объектами и прототипом предлагает нам метод `isPrototypeOf`. Допустим, у нас есть некий конструктор студентов, который на вход принимает имя студента и сохраняет его в поле `name` для каждого студента; хотим к каждому студенту привязать некоторый прототип. Мы помещаем этот прототип в поле `prototype`, в хранилище, и делаем его объектом на основе другого хранилища — конструктора `person` при помощи метода `create`. Далее, мы в наше хранилище добавляем специфичный для студентов метод `sleep` и восстанавливаем конструктор. Попробуем теперь создать нашего студента и воспользоваться для проверки связи между созданным студентом и его прототипом методом is`prototype`Of. Данный метод отвечает на вопрос: «Является ли объект прототипом для того объекта, который мы передаем в качестве аргументов?»

```
function Student(name) {
    this.name = name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.sleep = function () {};

Student.prototype.constructor = Student;

var billy = new Student('Billy');

Student.prototype.isPrototypeOf(billy); // true
```

Более того, данный метод позволяет инспектировать не только прямую связь, но и связь конечного объекта с одним из прототипов цепочки. Таким образом, он даст утвердительный ответ и на вопросы: «Является ли объект, который лежит в хранилище конструктора `person`, прототипом для `Billy`?» и «Является ли глобальный прототип для всех объектов прототипом для `Billy`?»

```
Person.prototype.isPrototypeOf(billy); // true

Object.prototype.isPrototypeOf(billy); // true
```

1.4.3. `instanceof`

Следующий способ инспектирования связей между объектами и конструкторами — специальный оператор `instanceof`. Он позволяет ответить на вопрос: «Является ли объект объектом определенного конструктора?» В данном случае мы создали нового студента `Billy` на основе конструктора Student и проверяем, является ли `Billy` объектом этого конструктора.

```
function Student(name) {
    this.name = name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.sleep = function () {};

Student.prototype.constructor = Student;

var billy = new Student('Billy');

billy instanceof Student; // true
```

Но этот оператор работает несколько сложнее, и он ответит, что `Billy` является и объектом конструктора `Person`, хотя напрямую это не так.

```
billy instanceof Person; // true

billy instanceof Object; // true
```

Но можно заметить, что хранилище, связанное с конструктором `Person`, лежит в цепочке прототипов до `Billy`. Если мы немножечко переформулируем то, как работает этот оператор, все встанет на свои места. Можно сказать, что он отвечает на вопрос: «Является ли `Billy` студентом?» или: «Является ли `Billy` личностью?» Более того, `Billy`, конечно же, является объектом — тоже правда.

Давайте разберем, как работает оператор `instanceof`. Например, как он проверяет, связаны ли между собой объект `Billy` с конструктором `Person`. Вначале он проверяет, является ли прототипом для `Billy` объект, который хранится в хранилище этого конструктора. И это не так. Тогда он проверяет следующую гипотезу: «А, может быть, прототипа у `Billy` совсем нет, и там хранится значение `null`?» Эта гипотеза также не подтверждается, и тогда он идет по цепочке прототипов. Он проверяет, является ли прототип прототипа `Billy` объектом, который хранится в хранилище конструктора `Person`. На этот раз гипотеза подтверждается, и в этом случае данный оператор отвечает нам `true`.

```
billy instanceof Person;
billy.__proto__ === Person.prototype;
// false -> Может, там null?
billy.__proto__ === null;
// false -> Идём дальше по цепочке
billy.__proto__.__proto__ === Person.prototype;
// true -> Возвращаем true
```

Если мы попробуем проинспектировать объект с самой короткой цепочкой прототипов, оператор `instanceof` возвращает нам `false`. Мы создаем объект и проверяем, что этот объект — это действительно объект, но оператор `instanceof` возвращает нам `false`. Если мы еще раз посмотрим, как работает этот оператор, все встанет на свои места. Итак, в первую очередь, он проверяет, является ли прототипом для нашего одинокого объекта глобальный прототип для всех объектов. Это не так. Тогда он проверяет: «Возможно, у нашего одинокого объекта совсем нет прототипа, и во внутреннем поле `prototype` лежит значение `null`?» В этом случае оператор `instanceof` при положительном подтверждении такой гипотезы возвращает нам `false`.

```
var foreverAlone = Object.create(null);

foreverAlone instanceof Object; // false

Object.create(null).__proto__ === Object.prototype;
// false -> Может, там null?

Object.create(null).__proto__ === null;
// true -> Так и есть, возвращаем false!
```

1.5. Решение проблемы дублирования кода в конструкторах

У нас есть конструктор студентов, конструктор преподавателей и конструктор личностей. Если мы посмотрим на конструкторы преподавателей и студентов, мы увидим, что они похожи и выполняют одинаковую работу: принимают в

качестве аргумента имя студента или преподавателя и сохраняют его в поле `name`.

```
function Student(name) {
    this.name = name;
}

function Lecturer(name) {
    this.name = name;
}

function Person() {
    this.type = 'human';
}
```

Нам бы хотелось избежать этого дублирования. Проще всего — вынести этот общий код в отдельный конструктор `Person`. Перенесем туда строчку, которая сохраняет имя студента или преподавателя в конструктор `Person`.

```
function Student(name) {
    this.name = name;
}

function Lecturer(name) {
    this.name = name;
}

function Person() {
    this.type = 'human';
}
```

Если мы сейчас попытаемся создать нового студента и передадим туда имя, мы не получим желаемого результата, так как мы забрали этот код из конструктора для студентов и переместили его в конструктор для личностей. Поэтому нам необходимо немножечко изменить реализацию конструктора студентов и добавить туда вызов конструктора личностей. Мы можем это сделать при помощи метода `call`. Вызываем наш конструктор личностей при помощи этого

метода и в качестве контекста передаем туда `this`, а в качестве второго аргумента передаем имя создаваемого нами студента.

```
function Person(name) {
    this.type = 'human';
    this.name = name;
}

function Student(name) {
    // this ссылается на новый объект студента
    Person.call(this, name);
}

var billy = new Student('Billy');

console.info(billy.name); // Billy
```

Благодаря тому, что в конструкторе `this` будет ссылаться на новый создаваемый объект, в нашем случае — на нового студента, мы вызовем конструктор `Person` с контекстом в виде этого студента, и таким образом передаваемое имя мы положим в поле name именно для студента. Все будет работать.

1.6. Вызов затеняемого метода в затеняющем

Допустим, у нас есть некоторый конструктор `Person`, конструктор личностей, который на вход принимает имя этих личностей и кладет в поле name. Также мы определяем прототип для всех личностей с методом `getName`, который возвращает это имя. Положим этот прототип в хранилище конструктора `Person`.

```
function Person(name) {
    this.name = name;
}

Person.prototype.getName = function () {
    return this.name;
}
```

Далее мы захотим использовать в нашей программе объекты другого типа и создадим конструктор для них. Прототипом для них мы сделаем объект на основе прототипа для личностей, но захотим изменить метод `getName`. Нам не хочется дублировать тот код, который есть в методе `getName` прототипа личностей, а лишь дополнить и добавлять к нему некую строку.
Вначале нам может показаться, что мы можем вызвать просто метод `getName` из прототипа `Person` внутри метода `getName` прототипа для студентов. Но в этом случае произойдет рекурсивный вызов того же самого метода. И интерпретатор нам скажет, что количество вызовов заполнило весь стек.

```
1  Student.prototype = Object.create(Person.prototype);
2
3  Student.prototype.getName = function () {
4      return 'Student ' + this.getName();
5  };
6
7  var billy = new Student('Billy');
8
9  billy.getName(); // RangeError: Maximum call stack size exceeded
```

Когда мы вызываем метод `getName` у созданного нами студента, например, `Billy`, внутри этого метода `this` будет ссылаться, собственно, на этого самого студента. Так как метода `getName` у самого `Billy` нет, он пойдет искать его в прототипе и найдет метод в прототипе, который хранится в хранилище конструктора `Student` в поле `Student.prototype`. Фактически метод будет вызывать сам себя.
Каким образом мы можем решить эту проблему? Самое простое — использовать другое название метода вместо эффекта затенения. В этом случае все будет работать. Мы будем вызывать метод с другим названием, например, getStudentName, заходить внутрь этого метода, `this` внутри него будет попрежнему ссылаться на объект `Student`, то есть на `Billy`, попробуем найти у `Billy` метод `getName`, не найдем его там, пройдем по цепочке прототипов в объект, который хранится в хранилище конструктора студентов Student.`prototype`. И там мы этого метода не найдем, проследуем дальше по цепочке прототипов и уже перейдем в хранилище, которое хранится в конструкторе `Person`. Там мы этот метод находим, спокойно его вызываем, возвращаем имя студента, добавляем к нему наш необходимый префикс, и все работает, как нужно.

```
function Person(name) {
    this.name = name;
}

Person.prototype.getName = function () {
    return this.name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.getStudentName = function () {
    return 'Student ' + this.getName();
};

var billy = new Student('Billy');

billy.getStudentName();
```

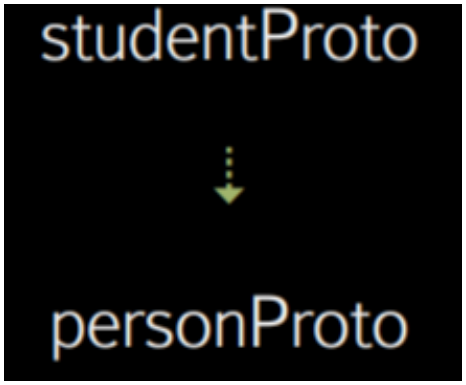
Более элегантным способом будет использование метода `call`. Мы можем напрямую в затеняющем методе вызывать затеняемый при помощи этого метода `call`, но передавать туда текущий контекст. А текущий контекст будет ссылаться на создаваемого объекта конструктора студентов, а именно на студента. Таким образом мы вызываем затеняемый метод от лица этого самого студента, получаем имя этого студента и добавляем к нему префикс `student`.

```
function Person(name) {
    this.type = 'human';
    this.name = name;
}

Person.prototype.getName = function () {
    return this.name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.getName = function () {
    return 'Student ' + Person.prototype.getName.call(this);
};
```

1.7. Сравнение трёх подходов к конструированию объектов: функции-конструкторы, метод `create`, «Классы»

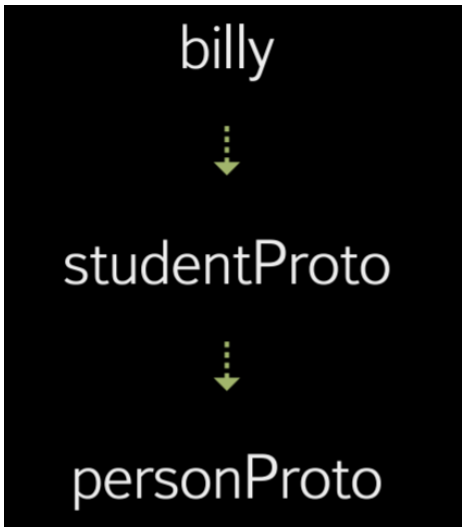
Для конструирования объектов и построения связей между ними достаточно лишь использовать только метод `create`.
Давайте снова попробуем создать конструктор для студентов. Для этого нам вновь понадобится ряд прототипов, выстроенных в цепочку. На этот раз мы воспользуемся только методом `create` и обычными объектами, так как любой прототип по факту — это обычный объект. Итак, для начала нам понадобится прототип личности. Создадим простой объект `personProto` и добавим туда метод `getName`, который будет возвращать имя нашей личности. На основе этого прототипа создадим более специфичный прототип уже для студентов. При помощи метода `create` мы свяжем два этих прототипа в цепочку. Далее мы можем расширить наш прототип для студентов уже специфичными для студентов методами, например, методом `sleep`.

```
var personProto = {
    getName: function () {
        return this.name;
    }
};
var studentProto =
 ↪  Object.create(personProto);

studentProto.sleep = function () {};
```



Далее на основе прототипа для студентов мы уже будем создавать студентов. Для этого снова воспользуемся методом `create`. Таким образом, благодаря этому методу мы встроим нашего студента в цепочку прототипов. И далее всё, что нам останется, это только дополнить нашего студента полями, которые необходимы каждому конкретному студенту с конкретными значениями. В данном случае мы присваиваем нашему студенту имя `Billy`.

```
var billy =
 ↪  Object.create(studentProto);

billy.name = 'Billy';
```



Благодаря такому подходу нам понадобилось значительно меньше строчек кода, чем в классическом, чтобы научить нашу программу создавать новые объекты студентов. Но и здесь мы можем произвести улучшение. Для этого нам понадобится ещё одна возможность метода `create`, а именно — данный метод принимает не один аргумент, а два. В качестве второго аргумента вы можете

передать необходимые поля с их начальными значениями и характеристиками, которые вы хотите видеть при создании объекта в итоговом объекте.

```
var apple = Object.create(fruit, {
    shape: { value: 'round', writable: false },
    color: { value: 'Green' },
    amount: { writable: true }
});

apple.amount = 'half';
```

Посмотрим, каким образом мы можем использовать данную возможность. Чтобы создавать наших студентов в одну строку, нам понадобится дополнительная функция-помощник, назовём её `create` и положим в наш прототип для студентов. Иногда такие функции называют фабриками — фабриками объектов. Данная функция будет на вход принимать в себя имя студента, а внутри себя вызывать метод `Object.create`, в качестве контекста передавать `this` и в качестве второго аргумента передавать набор полей, которые мы хотим видеть у студента, а именно: мы хотим у него видеть поле `name`, и мы сразу заполняем его тем, что нам приходит в фабрику. Таким образом мы можем создавать новых студентов в одну строчку, как и при классическом подходе при помощи оператора `new`. В данном случае мы просто вызываем наш метод `create`, который лежит в прототипе студента, передаём туда имя и получаем новый объект, нового студента с этим именем.

```
var personProto = {};

personProto.getName = function () { return this.name; }

var studentProto = Object.create(personProto);

studentProto.sleep = function () {};

studentProto.create = function (name) {
    return Object.create(this, {
        name: { value: name }
    });
}

var billy = studentProto.create('Billy');
```

Единого мнения, какой из этих подходов лучше, у разработчиков нет.
Также есть новая версия спецификаций, которая вводит новую синтаксическую конструкцию, а именно «классы». И вместо функций-конструкторов мы можем определить класс и на основе этого класса создавать новые объекты.

```
function Student(name) {
    this.name = name;
}

Student.prototype.getName = function () {
    return this.name
};

class Student {
    constructor(name) {
        this.name = name;
    }

    getName() {
        return this.name;
    }
}
```

Классы - это не более, чем обычные функции-конструкторы. У них также есть специальное поле `prototype` — хранилище для прототипов, для всех вновь создаваемых этим классом объектов. Также если мы посмотрим тип классов, то мы увидим, что это обычная функция.

```
class Student {
// ...
}

var billy = new Student('Billy');

billy.getName(); // Billy

Student.prototype.isPrototypeOf(billy); // true

typeof Student; // function
```

Таким образом, дискуссия трансформировалась в другую: что выбрать, классы или метод `Object.create`? Наиболее полно её раскрыл в статье разработчик Eric Elliott, и я рекомендую вам всем ознакомиться с ней.