ГЛАВА 9

Объекты и объектно-ориентированное программирование

Основы объектов JavaScript мы рассмотрели в главе 3, а теперь пришло время изучить их глубже.

Как и массивы, объекты в JavaScript — это *контейнеры*, которые называют *агрегатными* или комплексными *типами данных*. У объектов есть два основных отличия от массивов.

- Массивы содержат значения, индексированные в числовой форме; объекты содержат свойства, индексированные строкой или символом.
- Массивы упорядочены (элемент `arr[0]` всегда следует перед `arr[1]`); объекты не упорядочены (вы не можете гарантировать, что свойство `obj.a` расположено перед `obj.b`).

Эти различия носят довольно эзотерический (но важный) характер, поэтому давайте считать свойства тем, что делает объекты по настоящему особенными. *Свойство* (property) состоит из *ключа* (key) (строки или символа) и *значения* (value). Особенными объекты делает то, что вы можете обращаться к свойствам по их ключам.

Перебор свойств

Обычно, если нужно вывести содержимое некоторого контейнера (операция *перебора*), чаще всего используется массив, а не объект. Тем не менее объекты также являются контейнерами, и они обеспечивают перебор свойств; вам только нужно знать об особенностях и возможных сложностях.

Первое, что необходимо помнить о переборе свойств, — это то, что *порядок* не *гарантируется*. Проведя небольшой эксперимент, вы можете обнаружить, что свойства выводятся в том порядке, в котором они добавляются, и это справедливо для многих реализаций движка *почти всегда*. Тем не менее JavaScript не дает никаких гарантий этому, и изменение реализации движка в любой момент может

ликвидировать этот эффект. Поэтому не стоит полагаться на результат эксперимента и *никогда* не стоит подразумевать, что порядок свойств будет именно таков.

Будучи предупрежденными об этом, давайте теперь рассмотрим основные способы перебора свойств объекта.

Цикл `for...in`

Традиционным способом перебора свойств объекта является цикл `for...in`. Рассмотрим объект, у которого есть несколько строковых свойств и одно символьное свойство.

```
const SYM = Symbol();

const o = { a: 1, b: 2, c: 3, [SYM]: 4 };

for(let prop in o) {
    if(!o.hasOwnProperty(prop)) continue;
    console.log('${prop}: ${o[prop]}');
}
```

Все кажется довольно простым… кроме, вероятно, вполне резонного вопроса "Что делает `hasOwnProperty`?" Он ликвидирует опасность, связанную с циклом `for...in`, которая станет ясна далее в этой главе. Речь идет об унаследованных свойствах. В данном примере мы это могли бы опустить и не придать значения. Но, перебирая свойства объектов других типов (особенно объектов, производных от других), вы можете обнаружить свойства, которых не ожидали. Поэтому я рекомендую выработать привычку использовать для проверки метод `hasOwnProperty`. Вы скоро узнаете, почему это так важно, а также научитесь определять, когда его можно безболезненно (или желательно) опустить.

Обратите внимание, что цикл `for...in` не выводит значения свойств с символьными ключами.

Несмотря на то что с помощью цикла `for...in` можно выполнить перебор элементов массива, обычно это считается плохой идеей. Для массивов я рекомендую использовать обычный цикл `for` или `forEach`.

Метод `Object.keys`

Метод `Object.keys` позволяет получить все перечислимые строковые свойства объекта в виде массива.

```
const SYM = Symbol();

const o = { a: 1, b: 2, c: 3, [SYM]: 4 };

Object.keys(o).forEach(prop => console.log('${prop}: ${o[prop]}'));
```

Этот пример приводит к тому же результату, что и цикл `for...in` (здесь даже не нужно выполнять проверку с помощью метода `hasOwnProperty`). Это весьма удобно, когда нужно собрать ключи свойств объекта в виде массива. Например, это облегчает вывод всех свойств объекта, которые начинаются с символа *x*.

```
const o = { apple: 1, xochitl: 2, balloon: 3, guitar: 4, xylophone: 5, };

Object.keys(o)
    .filter(prop => prop.match(/^x/))
    .forEach(prop => console.log('${prop}: ${o[prop]}'));
```

Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП, Object-Oriented Programming) — старая добрая парадигма в информатике. Некоторые из концепций, которые мы теперь знаем как ООП, появились еще в 1950-х годах, но только после появления языков программирования Simula 67 и Smalltalk они обрели форму ООП.

Фундаментальная идея проста и интуитивно понятна: *объект* — это логически связанная коллекция данных и функций. Она призвана отразить наше понимание естественного мира. *Автомобиль* — это объект, у которого есть данные (марка, модель, количество дверей, идентификатор транспортного средства (VIN) и т.д.), а также функции (ускорение, переключение передач, открывание дверей, включение фар и т.д.). Кроме того, ООП позволяет думать о вещах абстрактно (*автомобиль*) и конкретно (*определенный* автомобиль).

Прежде чем продолжать, давайте рассмотрим базовую лексику ООП. Термин *класс* (class) описывает обобщенную сущность (*автомобиль*), а *экземпляр* (instance) (или *экземпляр объекта* (object instance)) — определенную сущность (*конкретный* автомобиль, такой как "мой автомобиль"). Одна часть функций (ускорение) является *методами* (method). Другая часть функций, связанных с классом, но не относящихся к конкретному экземпляру, является *методами класса* (например, "создание нового VIN" могло бы быть методом класса: это не имеет отношения к конкретному новому автомобилю и, конечно, мы не ожидаем, что у конкретного автомобиля будет возможность или способность создать новый, законный VIN). Когда экземпляр создается, выполняется его *конструктор* (constructor). Конструктор инициализирует экземпляр объекта.

ООП предоставляет нам также среду для иерархической категоризации классов. Например, мог бы существовать более общий класс *транспортного средства*. У транспортного средства может быть характеристика *дальности* (дистанция, которую он может пойти без дозаправки или перезарядки), но в отличие от автомобиля у него может не быть колес (например, у такого транспортного средства, как лодка, очевидно нет колес). Мы говорим, что транспортное средство — это *суперкласс* (superclass) автомобиля, а автомобиль — это *производный класс* (subclass) транспортного средства. У класса транспортного средства может быть несколько производных классов: автомобили, лодки, планеры, мотоциклы, велосипеды и т.д. У производных

классов, в свою очередь, могут быть следующие производные классы. Например, у производного класса лодки могут быть дальнейшие производные классы парусной яхты, гребной шлюпки, каноэ, буксира, моторной лодки и т.д.

В этой главе мы будем использовать пример автомобиля, поскольку это реальный объект, с которым все мы, очевидно связаны.

Создание класса и экземпляра

До ES6 создание классов в JavaScript было суетным и не интуитивно понятным делом. Теперь появился новый удобный синтаксис создания классов.

```
class Car {
    constructor() {
    }
}
```

Это создает новый класс по имени `Car`. Никаких его экземпляров (конкретных автомобилей) еще не создано, но теперь есть возможность сделать это. Чтобы создать конкретный автомобиль, мы используем ключевое слово `new`.

```
const car1 = new Car();
const car2 = new Car();
```

Теперь у нас есть два экземпляра класса `Car`. Прежде чем сделать класс `Car` более сложным, давайте рассмотрим оператор `instanceof`, который может сказать вам, является ли данный объект экземпляром данного класса.

```
car1 instanceof Car   // true
car1 instanceof Array // false
```

Из этого видно, что `car1` — экземпляр класса `Car`, а `Array` — нет.

Давайте сделаем класс `Car` немного интереснее. Придадим ему некие данные (марка, модель) и некие функции (переключение передач).

```
class Car {
    constructor(make, model) {
        this.make = make;
        this.model = model;
        this.userGears = ['P', 'N', 'R', 'D'];
        this.userGear = this.userGears[0];
    }
    shift(gear) {
        if(this.userGears.indexOf(gear) < 0)
            throw new Error('Ошибочная передача: ${gear}');
        this.userGear = gear;
    }
}
```

Здесь ключевое слово `this` используется по прямому назначению: для обращения к экземпляру, метод которого был вызван. Вы можете считать его знакоместом:

когда вы пишете свой класс, вероятно, абстрактный, ключевое слово this является знакоместом для *конкретного* экземпляра, который будет известен на момент вызова метода. Этот конструктор позволяет задать марку и модель автомобиля при его создании, а также установить некоторые стандартные значения: допустимые передачи (userGears) и текущую передачу (gear), которую мы инициализируем значением первой допустимой передачи. (Я решил назвать это *пользовательскими* передачами (user gears) потому, что если этот автомобиль оснащен автоматической коробкой передач, то, когда автомобиль будет находиться в движении, фактически использующаяся передача может отличаться от включенной пользователем.) Кроме конструктора (который вызывается неявно при создании нового объекта), мы также создали метод shift, позволяющий переключать передачу. Давайте рассмотрим это в действии.

```
const car1 = new Car("Tesla", "Model S");
const car2 = new Car("Mazda", "3i");
car1.shift('D');
car2.shift('R');
```

В этом примере, когда мы вызываем car1.shift('D'), переменная this связана с car1. Точно так же при вызове car2.shift('R') она связана с car2. Мы можем убедиться, что car1 находится в движении на передаче D (drive), а car2 сдает назад на передаче R (reverse).

```
> car1.userGear // "D"
> car2.userGear // "R"
```

Динамические свойства

То, что метод shift нашего класса Car предотвращает выбор недопустимой передачи по небрежности, может казаться очень умным ходом. Но эта защита ограничена, поскольку нет ничего, что помешало бы установить значение непосредственно: car1.userGear = 'X'. Большинство объектно-ориентированных языков идет на большие затраты, чтобы предоставить механизмы защиты от этого вида неправильного обращения, разрешая вам определять уровень доступа к методам и свойствам. В JavaScript такого механизма нет, за что его нередко и критикуют.

Динамические свойства[1] способны несколько сгладить этот недостаток. Они обладают семантикой свойств с функциональными возможностями методов. Давайте изменим наш класс Car так, чтобы использовать это в своих интересах.

```
class Car {
    constructor(make, model) {
        this.make = make;
        this.model = model;
        this._userGears = ['P', 'N', 'R', 'D'];
        this._userGear = this._userGears[0];
```

[1] Динамические свойства было бы правильнее называть *методами доступа к свойствам* (accessor properties), о которых мы узнаем больше в главе 21.

```
    }

    get userGear() { return this._userGear; }
    set userGear(value) {
        if(this._userGears.indexOf(value) < 0)
            throw new Error('Ошибочная передача: ${value}');
        this._userGear = value;
    }

    shift(gear) { this.userGear = gear; }
}
```

Проницательный читатель заметил, что мы не устранили проблему, поскольку значение `_userGear` все еще можно установить непосредственно: `car1._userGear = 'X'`. В этом примере мы используем "ограничение доступа для бедных" — свойства, имена которых начинаются с символа подчеркивания, мы считаем закрытыми. Эта защита сугубо в соглашении, позволяющем быстро просмотреть код и выявить свойства, к которыми вы не должны обращаться непосредственно.

Если вы действительно должны обеспечить конфиденциальность, то можете использовать экземпляр `WeakMap` (см. главу 10), который защищен областью видимости (если мы не будем использовать `WeakMap`, то наши закрытые свойства никогда не будут выходить из области видимости, даже если экземпляры, к которым они относятся, выйдут). Чтобы сделать основное текущее свойство передачи действительно закрытым, мы можем изменить свой класс `Car` так.

```
const Car = (function() {

    const carProps = new WeakMap();

    class Car {
        constructor(make, model) {
            this.make = make;
            this.model = model;
            this._userGears = ['P', 'N', 'R', 'D'];
            carProps.set(this, { userGear: this._userGears[0] });
        }

        get userGear() { return carProps.get(this).userGear; }
        set userGear(value) {
            if(this._userGears.indexOf(value) < 0)
                throw new Error('Ошибочная передача: ${value}');
            carProps.get(this).userGear = value;
        }

        shift(gear) { this.userGear = gear; }
    }

    return Car;
})();
```

Чтобы поместить наш `WeakMap` в замкнутое выражение, к которому нельзя обратиться извне, мы используем немедленно вызываемое функциональное выражение (см. главу 13). Теперь `WeakMap` может безопасно хранить любые свойства, к которым мы не хотим обращаться за пределами класса.

Существует и другой способ, который подразумевает использование символов для имен свойств; они также предоставляют некоторую защиту от случайного использования, но к символьным свойствам класса также можно обратиться, а значит, даже эту защиту можно обойти.

Классы как функции

До введения в ES6 ключевого слова `class` для создания класса приходилось создавать функцию, которая служила бы конструктором класса. Хотя синтаксис `class` намного более интуитивно понятен и прост, внутренний характер классов в JavaScript не изменился (ключевое слово `class` лишь обеспечивает немного более удобный синтаксис), поэтому важно понимать, что именно представляет собой класс в JavaScript.

В действительности класс — это только функция. В ES5 мы начали бы свой класс `Car` так.

```
function Car(make, model) {
    this.make = make;
    this.model = model;
    this._userGears = ['P', 'N', 'R', 'D'];
    this._userGear = this.userGears[0];
}
```

Мы все еще можем сделать это и в ES6 — результат будет тот же (до методов мы дойдем ниже). Мы можем проверить это, опробовав оба пути.

```
class Es6Car {}      // опустим конструктор для краткости
function Es5Car {}
> typeof Es6Car      // "function"
> typeof Es5Car      // "function"
```

Таким образом, ничего действительно нового в ES6 нет; у нас есть только некий новый удобный синтаксис.

Прототип

Когда говорят о методах, доступных в экземплярах класса, имеют в виду *прототип* (prototype) методов. Например, упоминая метод `shift`, доступный в экземплярах класса `Car`, вы имеете в виду прототип метода и зачастую можете встретить синтаксис `Car.prototype.shift`. (Точно так же функция `forEach` класса `Array` может выглядеть как `Array.prototype.forEach`.) Теперь пришло время фактически узнать, что такое прототип и как JavaScript осуществляет *динамический вызов* (dynamic dispatch), используя *цепь прототипов* (prototype chain).

Использование знака диеза (#) стало популярным соглашением для описания прототипов методов. Например, вы будете часто встречать `Car.prototype.shift`, записанный просто как `Car#shift`.

Каждая функция имеет специальное свойство `prototype`. (Вы можете изменить его для любой функции `f`, введя на консоли **`f.prototype`**.) Для обычных функций прототип не используется, но он критически важен для функций, которые действуют как конструкторы объектов.

В соответствии с соглашением имена конструкторов объектов (иначе — классов) всегда начинаются с заглавной буквы, например `Car`. Это соглашение — не догма, но многие анализаторы предупредят вас, если вы попытаетесь называть функцию с заглавной буквы или конструктор объекта — со строчной.

Свойство функции `prototype` становится важным, когда вы создаете новый экземпляр с использованием ключевого слова `new`: вновь созданный объект имеет доступ к свойству `prototype` его конструктора. Экземпляр объекта хранит его в своем свойстве `__proto__`.

Свойство `__proto__` считается внутренней частью JavaScript, как и любое свойство, заключенное между двойными символами подчеркивания. Используя эти свойства, можно сделать очень, очень много вреда. Иногда их можно использовать очень хитро и правильно, но пока у вас нет полного понимания JavaScript, я настоятельно рекомендую только просматривать (но не изменять) эти свойства.

В прототипе важнее всего механизм *динамического вызова* (термин “dispatch” — это синоним вызова метода). Когда вы пытаетесь получить доступ к свойству или методу объекта, если его не существует, JavaScript *проверяет прототип объекта*, чтобы убедиться, есть ли он там. Поскольку все экземпляры данного класса совместно используют один и тот же прототип, к свойству или методу, имеющемуся в прототипе, есть доступ для всех экземпляров этого класса.

Присвоение значения свойствам данных в прототипе класса обычно не выполняется. Дело в том что тогда значение этого свойства будет доступно для всех экземпляров класса. Однако если значение свойства *устанавливается* в каком-нибудь экземпляре, оно устанавливается именно в этом экземпляре, а не в прототипе, чтобы избежать путаницы и ошибок. Если экземпляры должны иметь начальные значения свойств данных, то лучше устанавливать их в конструкторе.

Обратите внимание, что определение метода или свойства в экземпляре переопределяет версию в прототипе; помните, что JavaScript сначала проверяет экземпляр, а только затем — прототип. Давайте рассмотрим все это на примере.

```
// определенный ранее класс Car с методом shift
const car1 = new Car();
const car2 = new Car();
car1.shift === Car.prototype.shift; // true
car1.shift('D');
car1.shift('d');                    // ошибка
car1.userGear;                      // 'D'
car1.shift === car2.shift           // true
car1.shift = function(gear) { this.userGear = gear.toUpperCase(); }
car1.shift === Car.prototype.shift; // false
car1.shift === car2.shift;          // false
car1.shift('d');
car1.userGear;                      // 'D'
```

В этом примере ясно показано, как JavaScript осуществляет динамический вызов. Первоначально у объекта `car1` нет метода `shift`, но при вызове `car1.shift('D')` JavaScript просматривает прототип для `car1` и находит метод с таким именем. Когда мы заменяем метод `shift` собственной версией, то у объекта `car1` и у его прототипа появляется метод с этим именем. Однако при вызове `car1.shift('d')`, будет вызван метод объекта `car1`, а не его прототипа.

Обычно в знании механики цепи прототипов и динамического вызова у вас не будет особой нужды, но все же может встретиться проблема, которая потребует их глубокого понимания. Поэтому, прежде чем продолжать, имеет смысл узнать детали.

Статические методы

Методы, которые мы рассматривали до сих пор, являлись *методами экземпляра* (instance method). Они предназначены для работы с конкретным экземпляром. Есть также *статические методы* (static method) (или *методы класса* (class method)), которые не относятся ни к какому конкретному экземпляру. В статическом методе переменная `this` привязана к самому классу, но в этом случае вместо нее рекомендуется использовать имя класса.

Статические методы используются для выполнения обобщенных задач, которые связаны с классом, а не с любым конкретным экземпляром. Давайте рассмотрим пример использования VIN автомобиля (идентификатор транспортного средства). Нет смысла позволять индивидуальному автомобилю создавать собственный VIN: что помешало бы автомобилю использовать такой же VIN, как и у другого автомобиля? Однако присвоение VIN является абстрактной концепцией, которая связана с идеей автомобиля вообще; следовательно, это кандидат в статические методы. Кроме того, статические методы зачастую используются для работы с несколькими транспортными средствами (объектами). Например, нам может понадобиться метод `areSimilar`, который возвращает `true`, если у двух автомобилей те же марка и модель, а метод `areSame`, возвращающий `true`, если у двух автомобилей один и тот же VIN. Давайте рассмотрим эти статические методы, реализованные для класса `Car`.

```
class Car {
    static getNextVin() {
        return Car.nextVin++; // мы могли бы также использовать
                              // this.nextVin++, но обращение к Car
                              // подчеркивает, что это статический метод
    }
    constructor(make, model) {
        this.make = make;
        this.model = model;
        this.vin = Car.getNextVin();
    }
    static areSimilar(car1, car2) {
        return car1.make===car2.make && car1.model===car2.model;
    }
    static areSame(car1, car2) {
        return car1.vin===car2.vin;
    }
}
Car.nextVin = 0;

const car1 = new Car("Tesla", "S");
const car2 = new Car("Mazda", "3");
const car3 = new Car("Mazda", "3");

car1.vin;  // 0
car2.vin;  // 1
car3.vin   // 2

Car.areSimilar(car1, car2);  // false
Car.areSimilar(car2, car3);  // true
Car.areSame(car2, car3);     // false
Car.areSame(car2, car2);     // true
```

Наследование

Рассматривая прототипы, мы уже встречали некий вид наследования: при создании экземпляра класса он наследовал все функции, находящиеся в прототипе класса. Но на этом дело не заканчивается: если метод не найден в прототипе объекта, проверяется прототип *прототипа*. Так получается *цепь прототипов*. JavaScript будет идти по цепи прототипов, пока не найдет тот прототип, который удовлетворяет запросу. Если такой прототип не будет найден, то все закончится ошибкой.

Это пригодится при создании иерархии классов. Мы уже упоминали, что автомобиль — это общий тип транспортного средства. Цепь прототипов позволяет располагать функции там, где им самое место. Например, у автомобиля мог бы быть метод `deployAirbags`. Мы могли бы сделать его методом обобщенного транспортного средства, но вы когда-либо видели лодку с подушками безопасности? С другой стороны, почти все транспортные средства могут перевозить пассажиров; таким образом, у транспортного средства мог бы быть метод `addPassenger` (который мог бы

сообщать об ошибке, если количество пассажиров превышено). Давайте посмотрим, как этот сценарий реализуется в коде JavaScript.

```
class Vehicle {
    constructor() {
        this.passengers = [];
        console.log("Транспортное средство создано");
    }
    addPassenger(p) {
        this.passengers.push(p);
    }
}

class Car extends Vehicle {
    constructor() {
        super();
        console.log("Автомобиль создан");
    }
    deployAirbags() {
        console.log("БАБАХ!!!");
    }
}
```

Первое нововведение, которое мы замечаем, — это ключевое слово `extends`; этот синтаксис указывает, что класс `Car` происходит от класса `Vehicle`. Второй новостью является вызов `super()`. Это специальная функция JavaScript, которая вызывает конструктор суперкласса. Для производных классов это обязательно; если вы опустите его, то получите ошибку.

Давайте рассмотрим этот пример в действии.

```
const v = new Vehicle();
v.addPassenger("Frank");
v.addPassenger("Judy");
v.passengers;                // ["Frank", "Judy"]
const c = new Car();
c.addPassenger("Alice");
c.addPassenger("Cameron");
c.passengers;                // ["Alice", "Cameron"]
v.deployAirbags();           // ошибка
c.deployAirbags();           // "БАБАХ!!!"
```

Обратите внимание, что мы можем вызвать метод `deployAirbags` с `c`, но не с `v`. Другими словами, наследование работает только в одном направлении. Экземпляры класса `Car` могут обращаться ко всем методам класса `Vehicle`, но не наоборот.

Полиморфизм

Термин *полиморфизм* (polymorphism) из лексикона объектно-ориентированных языков описывает ситуацию, когда экземпляр рассматривается как член не только

его собственного класса, но и любых суперклассов. На многих объектно-ориентированных языках полиморфизм — это нечто особенное, приносящее большую пользу ООП. Язык JavaScript не является типизированным, т.е. любой объект может быть использован в любом месте (хотя правильный результат не гарантирован). Таким образом, в некотором смысле у JavaScript есть абсолютный полиморфизм.

Код JavaScript, который вы пишете, довольно часто использует некую форму *утиной типизации* (duck typing). Эта методика исходит из выражения "Если это выглядит, как утка, плавает, как утка и крякает, как утка, то это, возможно, и есть утка". В нашем примере с классом `Car`, если у вас есть объект, обладающий методом `deployAirbags`, то вы могли бы резонно заключить, что это экземпляр класса `Car`. Это может быть правда, а может и нет, но попытка довольно хорошая.

В JavaScript предусмотрен оператор `instanceof`, который укажет вам, является ли объект экземпляром данного класса. Как ни удивительно, но до тех пор, пока вы не оперируете напрямую свойствами `prototype` и `__proto__`, этот оператор будет возвращать правильный результат.

```
class Motorcycle extends Vehicle {}
const c = new Car();
const m = new Motorcyle();
c instanceof Car;            // true
c instanceof Vehicle;        // true
m instanceof Car;            // false
m instanceof Motorcycle;     // true
m instanceof Vehicle;        // true
```

Все объекты в JavaScript являются экземплярами корневого класса `Object`. Таким образом, для любого объекта `o` выражение `o instanceof Object` будет истинным (если только вы явно не установите значение его свойства `__proto__`, чего следует избегать). С практической точки зрения в этом есть небольшой смысл, поскольку такая возможность позволяет создать ряд важных методов для всех объектов иерархии. В качестве примера можно привести метод `toString`, который будет рассмотрен ниже в этой главе.

Перебор свойств объектов (снова)

Мы уже видели, как можно перебрать свойства объекта в цикле `for...in`. Теперь, когда мы понимаем механизм наследования прототипов, мы можем полностью оценить использование метода `hasOwnProperty` при переборе свойств объекта. Для объекта `obj` и свойства `x` вызов `obj.hasOwnProperty(x)` возвратит `true`, если у `obj` будет свойство `x`, и `false`, если свойство `x` не определено или определено в цепи прототипов.

Если вы будете использовать классы ES6 так, как они задуманы, то свойства данных всегда будут определяться в экземплярах, а не в цепи прототипов. Однако,

поскольку нет ничего, что предотвратило бы добавление свойств непосредственно в прототип, всегда лучше использовать `hasOwnProperty`, чтобы удостовериться в этом. Рассмотрим пример.

```
class Super {
    constructor() {
        this.name = 'Super';
        this.isSuper = true;
    }
}

// это допустимо, но не желательно...
Super.prototype.sneaky = 'Не рекомендуется!';

class Sub extends Super {
    constructor() {
        super();
        this.name = 'Sub';
        this.isSub = true;
    }
}

const obj = new Sub();

for(let p in obj) {
    console.log('${p}: ${obj[p]}' +
         (obj.hasOwnProperty(p) ? '' : ' (унаследовано)'));
}
```

Если вы запустите эту программу, то увидите

```
name: Sub
isSuper: true
isSub: true
sneaky: Не рекомендуется! (унаследовано)
```

Свойства `name`, `isSuper` и `isSub` определяются в экземпляре, а не в цепи прототипов (обратите внимание, что свойства, объявленные в конструкторе суперкласса, присутствуют также в экземпляре производного класса). Свойство `sneaky`, напротив, было вручную добавлено в прототип суперкласса.

Вы можете избежать этой проблемы в целом, используя метод `Object.keys`, который включает только свойства, определенные в прототипе.

Строковое представление

Каждый объект в конечном счете происходит от класса `Object`. Таким образом, все методы, доступные в классе `Object`, стандартно доступны для всех объектов. Одним из этих методов является `toString`, возвращающий стандартное строковое

представление объекта. Стандартное поведение метода `toString` подразумевает возвращение строки `"[object Object]"`, что не особенно полезно.

Наличие метода `toString`, выводящего нечто описательное об объекте, может очень пригодиться при отладке, позволяя сразу получить важную информацию об объекте. Например, мы могли бы изменить свой класс `Car` так, чтобы его метод `toString` возвращал марку, модель и VIN.

```
class Car {
    toString() {
        return '${this.make} ${this.model}: ${this.vin}';
    }
    //...
```

Теперь вызов метода `toString` для экземпляра `Car` дает немного больше информации об объекте.

Множественное наследование, примеси и интерфейсы

Некоторые объектно-ориентированные языки поддерживают *множественное наследование* (multiple inheritance), когда у одного класса может быть два прямых суперкласса (в отличие от одного суперкласса, у которого, в свою очередь, есть один суперкласс). Множественное наследование создает риск *коллизий* (collision) или конфликтов. Таким образом, если нечто унаследовано от двух родителей и у обоих родителей есть метод `greet`, то от кого именно он будет унаследован производным классом? Во многих языках предпочитается одиночное наследование, при котором этой проблемы нет.

Но когда мы решаем реальные задачи, множественное наследование зачастую имеет смысл. Например, автомобили могли бы происходить как от транспортных средств, так и от "подлежащих страхованию" (вы можете застраховать и автомобиль, и дом, но дом, безусловно, — не транспортное средство). В языках, где не поддерживается множественное наследование, зачастую вводится концепция *интерфейса* (interface), чтобы обойти эту проблему. Класс (`Car`) может происходить только от одного родителя (`Vehicle`), но у него может быть несколько интерфейсов (`Insurable`, `Container` и т.д.).

JavaScript — интересный гибрид. Технически это язык одиночного наследования, поскольку поиск по цепи прототипов не распространяется на несколько родителей, но он предоставляет пути, которые иногда превосходят и множественное наследование, и интерфейсы (а иногда — нет).

Основной механизм решения проблемы множественного наследования — это концепция *примеси* (mixin). Примесь позволяет "подмешивать" функциональные возможности по мере необходимости. Поскольку JavaScript позволяет чрезвычайно много и без контроля типов, вы можете подмешать почти любые функции к любому объекту в любое время.

Давайте создадим примесь "страхуемый", которую мы могли бы применить к автомобилям. Мы не будем усложнять пример, но в дополнение к примеси страхования необходимо создать класс `InsurancePolicy`. Примесь страхования нуждается в методах `addInsurancePolicy`, `getInsurancePolicy` и (для удобства) `isInsured`. Давайте рассмотрим, как это могло бы работать.

```
class InsurancePolicy() {}
function makeInsurable(o) {
    o.addInsurancePolicy = function(p) { this.insurancePolicy = p; }
    o.getInsurancePolicy = function() { return this.insurancePolicy; }
    o.isInsured = function() { return !!this.insurancePolicy; }
}
```

Теперь мы можем сделать любой объект подлежащим страхованию. Так как мы собираемся сделать подлежащим страхованию класс `Car`? Ваша первая мысль могла бы быть такой.

```
makeInsurable(Car);
```

Но вы были бы неприятно удивлены.

```
const car1 = new Car();
car1.addInsurancePolicy(new InsurancePolicy()); // ошибка
```

Если вы подумали "Конечно, ведь `addInsurancePolicy` не находится в цепи прототипов", то будете совершенно правы. Делать класс `Car` подлежащим страхованию вообще плохая идея. Кроме того, это вообще не имеет смысла: абстрактная концепция автомобиля не подлежит страхованию, но конкретный автомобиль — подлежит. Таким образом, наше следующее решение могло бы быть таким.

```
const car1 = new Car();
makeInsurable(car1);
car1.addInsurancePolicy(new InsurancePolicy()); // работает
```

Это работает, но теперь нужно не забыть вызывать функцию `makeInsurable` для каждого создаваемого нами автомобиля. Мы могли бы добавить этот вызов в конструктор `Car`, но тогда мы продублируем эту функцию для каждого созданного автомобиля. К счастью, решение простое.

```
makeInsurable(Car.prototype);
const car1 = new Car();
car1.addInsurancePolicy(new InsurancePolicy()); // работает
```

Теперь это выглядит так, как будто наши методы всегда были частью класса `Car`. И с точки зрения JavaScript *так и есть*. С точки зрения разработчика мы облегчили поддержку этих двух важных классов. Группа разработчиков автомобилей создает и обслуживает класс `Car`, а группа страхования занимается классом `InsurancePolicy` и примесью `makeInsurable`. В результате место для пересечения двух групп все таки имеется, но это куда лучше, чем когда все работают над одним гигантским классом `Car`.

Примеси не устраняют проблему коллизий: если бы по каким-то причинам страховая группа должна была бы создать в своей примеси метод `shift`, то это нарушило бы класс `Car`. Кроме того, мы не можем использовать `instanceof` для выявления объектов, которые допускают страхование: в лучшем случае мы можем рассчитывать на утиную типизацию (если у объекта есть метод `addInsurancePolicy`, он, вероятно, подлежит страхованию).

Мы можем сгладить некоторые из этих проблем, используя символы. Скажем, страховая группа постоянно добавляет очень обобщенные методы, которые конфликтуют с методами класса `Car`. Вы могли бы попросить их использовать для всех своих ключей символы. Теперь их примесь будет выглядеть следующим образом.

```
class InsurancePolicy() {}
const ADD_POLICY = Symbol();
const GET_POLICY = Symbol();
const IS_INSURED = Symbol();
const _POLICY = Symbol();
function makeInsurable(o) {
    o[ADD_POLICY] = function(p) { this[_POLICY] = p; }
    o[GET_POLICY] = function() { return this[_POLICY]; }
    o[IS_INSURED] = function() { return !!this[_POLICY]; }
}
```

Поскольку символы уникальны, это гарантирует, что примесь никогда не будет конфликтовать с существующими функциями класса `Car`. В использовании это может быть немного неуклюже, но намного безопаснее. Компромиссный подход, возможно, подразумевал бы использование обычных строк для методов, а символов (таких, как `_POLICY`) для свойств данных.

Заключение

Объектно-ориентированное программирование — это чрезвычайно популярная парадигма и на то есть серьезные причины. Оно обеспечивает организацию и инкапсуляцию кода для решения многих реальных задач, что облегчает поддержку, отладку и исправление ошибок. Реализацию ООП в JavaScript серьезно критикуют — некоторые доходят до утверждения, что JavaScript даже не соответствует определению объектно-ориентированного языка (обычно из-за нехватки контроля доступа к данным). Этот аргумент заслуживает внимания, но как только вы привыкнете к тому, как в JavaScript реализовано ООП, вы найдете этот язык весьма гибким и мощным. Он позволяет делать такие вещи, на которые другие объектно-ориентированные языки не способны.