

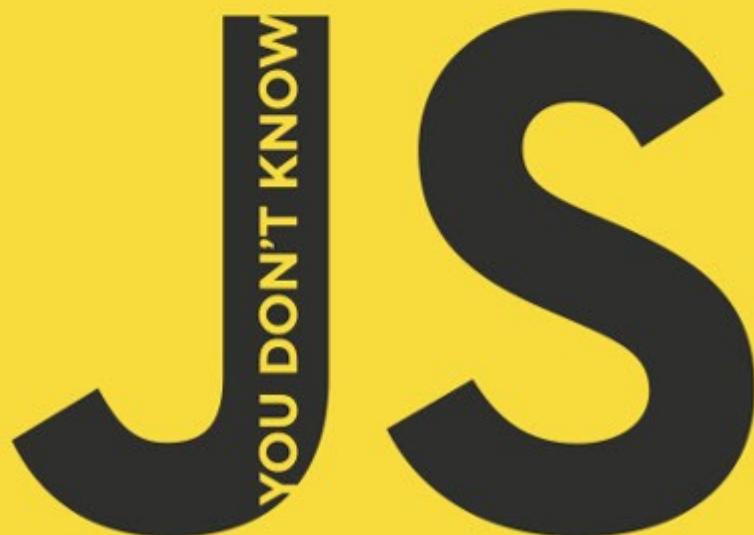
O'REILLY®

"The `this` keyword and prototypes are pivotal, because they are foundational to doing real programming with JavaScript."

-NICK BERARDI, Senior Consultant, RDA Corporation

KYLE SIMPSON

this & OBJECT PROTOTYPES



You Don't Know JS: *this* & Object Prototypes

Table of Contents

- Foreword
- Preface
- Chapter 1: *this* Or That?
 - Why *this*?
 - Confusions
 - What's *this*?
- Chapter 2: *this* All Makes Sense Now!
 - Call-site
 - Nothing But Rules
 - Everything In Order
 - Binding Exceptions
 - Lexical *this*
- Chapter 3: Objects
 - Syntax
 - Type
 - Contents
 - Iteration
- Chapter 4: Mixing (Up) "Class" Objects
 - Class Theory
 - Class Mechanics
 - Class Inheritance
 - Mixins
- Chapter 5: Prototypes
 - `[[Prototype]]`
 - "Class"
 - "(Prototypal) Inheritance"
 - Object Links
- Chapter 6: Behavior Delegation
 - Towards Delegation-Oriented Design
 - Classes vs. Objects
 - Simpler Design
 - Nicer Syntax
 - Introspection
- Appendix A: ES6 `class`
- Appendix B: Acknowledgments

Foreword

While reading this book in preparation for writing this foreword, I was forced to reflect on how I learned JavaScript and how much it has changed over the last 15 years that I have been programming and developing with it.

When I started using JavaScript 15 years ago, the practice of using non-HTML technologies such as CSS and JS in your web pages was called DHTML or Dynamic HTML. Back then, the usefulness of JavaScript varied greatly and seemed to be tilted toward adding animated snowflakes to your web pages or dynamic clocks that told the time in the status bar. Suffice it to say, I didn't really pay much attention to JavaScript in the early part of my career because of the novelty of the implementations that I often found on the Internet.

It wasn't until 2005 that I first rediscovered JavaScript as a real programming language that I needed to pay closer attention to. After digging into the first beta release of Google Maps, I was hooked on the potential it had. At the time, Google Maps was a first-of-its-kind application -- it allowed you to move a map around with your mouse, zoom in and out, and make server requests without reloading the page -- all with JavaScript. It seemed like magic!

When anything seems like magic, it is usually a good indication you are at the dawn of a new way of doing things. And boy, was I not wrong -- fast-forwarding to today, I would say that JavaScript is one of the primary languages I use for both client- and server-side programming, and I wouldn't have it any other way.

One of my regrets as I look over the past 15 years is that I didn't give JavaScript more of a chance before 2005, or more accurately, that I lacked the foresight to see JavaScript as a true programming language that is just as useful as C++, C#, Java, and many others.

If I had this *You Don't Know JS* series of books at the start of my career, my career history would look much different than it does today. And that is one of the things I love about this series: it explains JS at a level that builds your understanding as you go through the series, but in a fun and informative way.

this & Object Prototypes is a wonderful continuation to the series. It does a great and natural job of building on the prior book, *Scope & Closures*, and extending that knowledge to a very important part of the JS language, the `this` keyword and prototypes. These two simple things are pivotal for what you will learn in the future books, because they are foundational to doing real programming with JavaScript. The concept of how to create objects, relate them, and extend them to represent things in your application is necessary to create large and complex applications in JavaScript. And without them, creating complex applications (such as Google Maps) wouldn't be possible in JavaScript.

I would say that the vast majority of web developers probably have never built a JavaScript object and just treat the language as event-binding glue between buttons and AJAX requests. I was in that camp at a point in my career, but after I learned how to master prototypes and create objects in JavaScript, a world of possibilities opened up for me. If you fall into the category of just creating event-binding glue code, this book is a must-read; if you just need a refresher, this book will be a go-to resource for you. Either way, you will not be disappointed. Trust me!

Nick Berardi

nickberardi.com, @nberardi

Preface

I'm sure you noticed, but "JS" in the book series title is not an abbreviation for words used to curse about JavaScript, though cursing at the language's quirks is something we can probably all identify with!

From the earliest days of the web, JavaScript has been a foundational technology that drives interactive experience around the content we consume. While flickering mouse trails and annoying pop-up prompts may be where JavaScript started, nearly 2 decades later, the technology and capability of JavaScript has grown many orders of magnitude, and few doubt its importance at the heart of the world's most widely available software platform: the web.

But as a language, it has perpetually been a target for a great deal of criticism, owing partly to its heritage but even more to its design philosophy. Even the name evokes, as Brendan Eich once put it, "dumb kid brother" status next to its more mature older brother "Java". But the name is merely an accident of politics and marketing. The two languages are vastly different in many important ways. "JavaScript" is as related to "Java" as "Carnival" is to "Car".

Because JavaScript borrows concepts and syntax idioms from several languages, including proud C-style procedural roots as well as subtle, less obvious Scheme/Lisp-style functional roots, it is exceedingly approachable to a broad audience of developers, even those with just little to no programming experience. The "Hello World" of JavaScript is so simple that the language is inviting and easy to get comfortable with in early exposure.

While JavaScript is perhaps one of the easiest languages to get up and running with, its eccentricities make solid mastery of the language a vastly less common occurrence than in many other languages. Where it takes a pretty in-depth knowledge of a language like C or C++ to write a full-scale program, full-scale production JavaScript can, and often does, barely scratch the surface of what the language can do.

Sophisticated concepts which are deeply rooted into the language tend instead to surface themselves in *seemingly* simplistic ways, such as passing around functions as callbacks, which encourages the JavaScript developer to just use the language as-is and not worry too much about what's going on under the hood.

It is simultaneously a simple, easy-to-use language that has broad appeal, and a complex and nuanced collection of language mechanics which without careful study will elude *true understanding* even for the most seasoned of JavaScript developers.

Therein lies the paradox of JavaScript, the Achilles' Heel of the language, the challenge we are presently addressing. Because JavaScript *can* be used without understanding, the understanding of the language is often never attained.

‣ Mission

If at every point that you encounter a surprise or frustration in JavaScript, your response is to add it to the blacklist, as some are accustomed to doing, you soon will be relegated to a hollow shell of the richness of JavaScript.

While this subset has been famously dubbed "The Good Parts", I would implore you, dear reader, to instead consider it the "The Easy Parts", "The Safe Parts", or even "The Incomplete Parts".

This *You Don't Know JavaScript* book series offers a contrary challenge: learn and deeply understand *all* of JavaScript, even and especially "The Tough Parts".

Here, we address head on the tendency of JS developers to learn "just enough" to get by, without ever forcing themselves to learn exactly how and why the language behaves the way it does. Furthermore, we eschew the common advice to *retreat* when the road gets rough.

I am not content, nor should you be, at stopping once something *just works*, and not really knowing *why*. I gently challenge you to journey down that bumpy "road less traveled" and embrace all that JavaScript is and can do. With that knowledge, no technique, no framework, no popular buzzword acronym of the week, will be beyond your understanding.

These books each take on specific core parts of the language which are most commonly misunderstood or under-understood, and dive very deep and exhaustively into them. You should come away from reading with a firm confidence in your understanding, not just of the theoretical, but the practical "what you need to know" bits.

The JavaScript you know *right now* is probably *parts* handed down to you by others who've been burned by incomplete understanding. *That* JavaScript is but a shadow of the true language. You don't *really* know JavaScript, *yet*, but if you dig into this series, you *will*. Read on, my friends. JavaScript awaits you.

Summary

JavaScript is awesome. It's easy to learn partially, and much harder to learn completely (or even *sufficiently*). When developers encounter confusion, they usually blame the language instead of their lack of understanding. These books aim to fix that, inspiring a strong appreciation for the language you can now, and *should*, deeply *know*.

Note: Many of the examples in this book assume modern (and future-reaching) JavaScript engine environments, such as ES6. Some code may not work as described if run in older (pre-ES6) engines.

Глава 1: this (тут) или That (там)?

Одним из наиболее запутанных механизмов в Javascript является ключевое слово `this`. Это специальное ключевое слово идентификатор, которое автоматически определяется внутри области видимости каждой функции, но к чему именно оно относится сбивает с толку даже опытных Javascript-разработчиков.

Любая достаточно продвинутая технология неотличима от магии. -- Артур Си. Клэрк

Механизм `this` Javascript на самом деле не такой уж и продвинутый, но разработчики часто перефразируют эту цитату вставив "сложный" или "сбивающий с толку", и совершенно понятно, что без четкого понимания это может казаться совершенно магическим в вашем понимании.

Примечание: Слово "`this`" — это достаточно распространенное местоимение в общих беседах. Поэтому, может быть очень сложно, особенно на словах, определить используем мы "`this`" как местоимение или же используем его, чтобы ссылаться на данное ключевое слово. Для ясности, я всегда буду использовать `this` для ссылки на специальное ключевое слово, а "`this`" или `this` или `this` в остальных случаях.

’ Зачем нужен `this`?

Раз механизм `this` такой запутанный даже для опытных JavaScript-разработчиков, можно задаться вопросом, а точно ли он полезный? Может у него больше недостатков, чем достоинств? Перед тем, как перейти к тому как он работает, мы должны проанализировать зачем он нужен.

Давайте попытаемся проиллюстрировать мотивацию и полезность механизма `this`:

```
function identify() {
    return this.name.toUpperCase();
}

function speak() {
    var greeting = "Hello, I'm " + identify.call( this );
    console.log( greeting );
}

var me = {
    name: "Kyle"
};

var you = {
    name: "Reader"
};

identify.call( me ); // KYLE
identify.call( you ); // READER
```

```
speak.call( me ); // Hello, I'm KYLE  
speak.call( you ); // Hello, I'm READER
```

Если то, как работает этот фрагмент кода путает вас, не волнуйтесь! Мы скоро вернемся к этому. Просто отложите ваши вопросы в сторону, чтобы мы могли более четко взглянуть на то, зачем это нужно.

Этот фрагмент кода позволяет функциям `identify()` и `speak()` быть переиспользованными с разными объектами контекста (`me` и `you`), а не требовать новой версии функции для каждого объекта.

Вместо того, чтобы полагаться на `this`, вы могли бы явно передать *объект контекста* функциям `identify()` и `speak()`.

```
function identify(context) {  
    return context.name.toUpperCase();  
}  
  
function speak(context) {  
    var greeting = "Hello, I'm " + identify( context );  
    console.log( greeting );  
}  
  
identify( you ); // READER  
speak( me ); // Hello, I'm KYLE
```

Однако, механизм `this` предоставляет более элегантный путь, неявно "передавая" ссылку на объект, что приводит к чистому дизайну API и облегчению повторного переиспользования.

Чем сложнее будет используемый вами паттерн, тем более ясно вы увидите, что указание контекста явным параметром часто запутаннее, чем неявное указание контекста `this`. Когда мы изучим объекты и прототипы, вы увидите полезность коллекции функций, которые способны автоматически ссылаться на правильный объект контекста.

› Заблуждения

Мы скоро объясним как `this` на самом деле работает, но сначала мы должны рассеять несколько заблуждений о том, как он на самом деле не работает.

Имя "`this`" создает заблуждение, когда разработчики пытаются думать о нем слишком буквально. Есть два часто предполагаемых значения, но оба являются неверными.

› Сама функция

Первый общий соблазн это предполагать, что `this` ссылается на саму функцию. Это, как минимум, резонное грамматическое заключение.

Но зачем вы бы хотели ссылаться на функцию из неё же? Наиболее распространенной причиной может быть такая вещь как рекурсия(вызов функции внутри себя) или чтобы назначить обработчик события, который сработает при вызове функции.

Разработчики новых механизмов JavaScript часто думают, что ссылка на функцию как на объект(все функции в JavaScript являются объектами!) позволяет хранить состояния(значения в свойствах) между вызовами функций. Хотя это, конечно, возможно, но это имеет некоторые ограничения в использовании, остаток книги будет повествовать о многих других шаблонах для лучшего хранения состояния, чем объект функции.

Но для начала мы используем этот шаблон, чтобы проиллюстрировать как `this` не дает функции получить ссылку на саму себя, как мы могли бы предположить.

Рассмотрим следующий код, где мы попытаемся отследить сколько раз функция (`foo`) была вызвана:

```
function foo(num) {
    console.log("foo: " + num);

    // Отслеживаем сколько раз `foo` была вызвана
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo(i);
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// Сколько раз была вызвана `foo`?
console.log(foo.count); // 0 -- WTF?
```

`foo.count` до сих пор равен 0, даже несмотря на то, что 4 инструкции `console.log` очевидно показывают, что `foo(..)` на самом деле была вызвана 4 раза. Разочарование происходит от слишком буквального толкования того, что означает `this` (в `this.count++`).

Когда код выполняет команду `foo.count = 0`, он на самом деле добавляет свойство `count` в объект функции `foo`. Но для ссылки `this.count` внутри функции `this` фактически не указывает на тот же объект функции, и несмотря на то, что имена свойств одинаковые, это разные объекты, вот тут то и начинается неразбериха.

Примечание: ответственный разработчик в этом месте должен спросить: "Если я увеличил свойство `count`, но оно не то, которое я ожидал, то какое `count` было мной увеличено?". На самом деле, если он копнет глубже, он обнаружит что случайно создал глобальную

переменную count(смотрите в главе 2 как это произошло!), а её текущим значением является NaN. Конечно, после того, как он определит это, у него появится совсем другой ряд вопросов: "почему она стала глобальной и почему она имеет значение NaN, вместо правильного значения счетчика?". (см. главу 2).

Вместо того, чтобы остановиться на этом месте и копнуть глубже, чтобы узнать почему ссылка this не ведет себя как ожидалось, большинство разработчиков просто откладывают проблему целиком и ищут другие решения, например, создают другой объект для хранения свойства count:

```
function foo(num) {
    console.log("foo: " + num);

    // отслеживаем сколько раз вызывалась `foo`
    data.count++;
}

var data = {
    count: 0
};

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo(i);
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// сколько раз вызывалась `foo`?
console.log(data.count); // 4
```

Хоть это и верно, что этот подход "решает" проблему, к сожалению, это просто игнорирование реальной проблемы — недостатка понимания того, что значит this и как он работает и вместо этого возвращение в зону комфорта более простого механизма: области видимости.

Примечание: Области видимости - замечательный и полезный механизм. Я не против использования их любым способом(см. книгу "Области видимости и замыкания" из этой серии книг). Но постоянно гадать, как использовать this, и, как правило, ошибаться — не лучшая причина возвращаться к областям видимости и никогда не узнать почему this ускользает от вас.

Для ссылки на объект функции изнутри этой функции, this самого по себе обычно бывает недостаточно. Вам обычно нужна ссылка на объект функции через лексический идентификатор (переменную), который указывает на него.

Рассмотрим эти 2 функции:

```

function foo() {
    foo.count = 4; // `foo` ссылается на саму себя
}

setTimeout( function(){
    // анонимная функция (без имени), не может
    // ссылаться на себя
}, 10 );

```

В первой функции вызывалась "именованная функция", foo — это ссылка, которая может быть использована для ссылки на функцию из самой себя.

Но во втором примере функция обратного вызова, передаваемая в `setTimeout(..)`, не имела имени идентификатора (так называемая "анонимная функция"), так что у неё нет правильного пути чтобы обратиться к её объекту.

Примечание: Старомодная, но ныне устаревшая и неиспользуемая ссылка `arguments.callee` внутри функции также указывает на объект функции, которая в данный момент выполняется. Эта ссылка обычно используется как возможность получить объект анонимной функции изнутри этой функции. Лучший подход, однако, состоит в том, чтобы избежать использования анонимных функций, по крайней мере тех, которые требуют обращения к себе изнутри, и вместо них использовать именованные функции. `arguments.callee` устарела и не должна использоваться.

Таким образом, другое решение нашего примера — это использовать идентификатор `foo` как ссылку на объект функции в каждом месте и вообще не использовать `this`, и это *работает*:

```

function foo(num) {
    console.log( "foo: " + num );

    // следим, сколько раз вызывается функция
    foo.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// сколько раз `foo` была вызвана?
console.log( foo.count ); // 4

```

Однако, этот подход также является уклонением от фактического понимания `this`, и полностью зависит от области видимости переменной `foo`.

Еще один путь решения проблемы - это заставить `this` действительно указывать на объект функции `foo`:

```
function foo(num) {
    console.log( "foo: " + num );

    // следим, сколько раз вызывается функция
    // Заметьте: `this` теперь действительно ссылается на `foo`, это основано на том,
    // как `foo` вызывается (см. ниже)
    this.count++;

}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        // используя `call(..)` мы гарантируем что `this`
        // ссылается на объект функции (`foo`) изнутри
        foo.call( foo, i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// сколько раз `foo` была вызвана?
console.log( foo.count ); // 4
```

Вместо избегания `this`, мы воспользовались им. Мы отведем немного времени на то, чтобы объяснить более детально как такие методы работают, так что не волнуйтесь если вы до сих пор недоумеваете как это работает!

Это область видимости функции

Следующее большое общее заблуждение касательно того, на что указывает `this` - это то, что он каким-то образом ссылается на область видимости функции. Это очень сложный вопрос, потому что с одной стороны так и есть, но с другой это совершенно не так.

Для ясности, `this`, в любом случае, не ссылается на область видимости функции. Это правда, что внутри области видимости имеет вид объекта со свойствами для каждого определенного значения. Но "объект" области видимости не доступен в JavaScript коде. Это внутренняя часть механизма реализации языка (интерпретатора).

Рассмотрим код, который пытается (и безуспешно!) перейти границу и использовать `this` неявно ссылаясь на область видимости функции:

```
function foo() {
    var a = 2;
    this.bar();
}

function bar() {
    console.log( this.a );
}

foo(); //undefined
```

В этом коде содержится более одной ошибки. Хотя он может казаться надуманным, код который вы видите — это фрагмент из реального практического кода, которым обменивались в публичных форумах сообщества. Это замечательная (если не печальная) иллюстрация того, насколько ошибочным может быть предположение о `this`.

Во-первых, попытка ссылаться на функцию `bar()` как `this.bar()`. Это почти наверняка случайность, что это работает, но мы коротко объясним как это работает позже. Наиболее естественным путем вызвать `bar()` было бы опустить предшествующий `this.` и просто сделать ссылку на идентификатор.

Однако, разработчик, который писал этот код, пытался использовать `this`, чтобы создать мост между областями видимости `foo()` и `bar()` так, чтобы `bar()` получила доступ к переменной `a` внутри области видимости `foo()`. **Не всякий мост возможен.** Вы не можете использовать ссылку `this`, чтобы найти что-нибудь в области видимости. Это невозможно.

Каждый раз, когда вы чувствуете, что вы смешиваете поиски в области видимости с `this`, напоминайте себе : *это не мост*.

Что же такое `this`?

Оставив ошибочные предположения, давайте обратим наше внимание на то, как механизм `this` действительно работает.

Мы ранее сказали, что `this` привязывается не во время написания функции, а во время её вызова. Это вытекает из контекста, который основывается на обстоятельствах вызова функции. Привязка `this` не имеет ничего общего с определением функции, но зависит от того при каких условиях функция была вызвана.

Когда функция вызывается, создается запись активации, также известная как контекст вызова. Эта запись содержит информацию о том, откуда функция была вызвана (стэк вызова), как функция была вызвана, какие параметры были в неё переданы и т.д. Одним из свойств этой записи является ссылка `this`, которая будет использоваться на протяжении выполнения этой функции.

В следующей главе мы научимся находить **место вызова** функции, чтобы определить как оно связано с определением `this`

Обзор (TL;DR)

Определение `this` - постоянный источник заблуждений для JavaScript разработчиков, которые не уделяют времени на изучение того, как этот механизм в действительности работает. Гадать, методом проб и ошибок, и слепо копировать код из StackOverflow - неэффективный и неправильный путь использовать этот важный механизм `this`.

Чтобы понять что такое `this`, вам сначала нужно понять чем `this` не является, несмотря на любые предположения или заблуждения, которые могут тянуть вас вниз. `this` — это не ссылка функции на саму себя и это не ссылка на область видимости функции.

В действительности `this` — это привязка, которая создается во время вызова функции, и на что она ссылается определяется тем, где и при каких условиях функция была вызвана.

Chapter 4: Mixing (Up) "Class" Objects

Following our exploration of objects from the previous chapter, it's natural that we now turn our attention to "object oriented (OO) programming", with "classes". We'll first look at "class orientation" as a design pattern, before examining the mechanics of "classes": "instantiation", "inheritance" and "(relative) polymorphism".

We'll see that these concepts don't really map very naturally to the object mechanism in JS, and the lengths (mixins, etc.) many JavaScript developers go to overcome such challenges.

Note: This chapter spends quite a bit of time (the first half!) on heavy "objected oriented programming" theory. We eventually relate these ideas to real concrete JavaScript code in the second half, when we talk about "Mixins". But there's a lot of concept and pseudo-code to wade through first, so don't get lost -- just stick with it!

’ Class Theory

"Class/Inheritance" describes a certain form of code organization and architecture -- a way of modeling real world problem domains in our software.

OO or class oriented programming stresses that data intrinsically has associated behavior (of course, different depending on the type and nature of the data!) that operates on it, so proper design is to package up (aka, encapsulate) the data and the behavior together. This is sometimes called "data structures" in formal computer science.

For example, a series of characters that represents a word or phrase is usually called a "string". The characters are the data. But you almost never just care about the data, you usually want to *do things* with the data, so the behaviors that can apply to that data (calculating its length, appending data, searching, etc.) are all designed as methods of a string class.

Any given string is just an instance of this class, which means that it's a neatly collected packaging of both the character data and the functionality we can perform on it.

Classes also imply a way of *classifying* a certain data structure. The way we do this is to think about any given structure as a specific variation of a more general base definition.

Let's explore this classification process by looking at a commonly cited example. A *car* can be described as a specific implementation of a more general "class" of thing, called a *vehicle*.

We model this relationship in software with classes by defining a *Vehicle* class and a *car* class.

The definition of *Vehicle* might include things like propulsion (engines, etc.), the ability to carry people, etc., which would all be the behaviors. What we define in *Vehicle* is all the stuff that is common to all (or most of) the different types of vehicles (the "planes, trains, and automobiles").

It might not make sense in our software to re-define the basic essence of "ability to carry people" over and over again for each different type of vehicle. Instead, we define that capability once in *Vehicle*, and then when we define *car*, we simply indicate that it "inherits" (or "extends") the base definition from *Vehicle*. The definition of *car* is said to specialize the general *Vehicle* definition.

While `Vehicle` and `car` collectively define the behavior by way of methods, the data in an instance would be things like the unique VIN of a specific car, etc.

And thus, classes, inheritance, and instantiation emerge.

Another key concept with classes is "polymorphism", which describes the idea that a general behavior from a parent class can be overridden in a child class to give it more specifics. In fact, relative polymorphism lets us reference the base behavior from the overridden behavior.

Class theory strongly suggests that a parent class and a child class share the same method name for a certain behavior, so that the child overrides the parent (differentially). As we'll see later, doing so in your JavaScript code is opting into frustration and code brittleness.

”Class” Design Pattern

You may never have thought about classes as a "design pattern", since it's most common to see discussion of popular "OO Design Patterns", like "Iterator", "Observer", "Factory", "Singleton", etc. As presented this way, it's almost an assumption that OO classes are the lower-level mechanics by which we implement all (higher level) design patterns, as if OO is a given foundation for *all* (proper) code.

Depending on your level of formal education in programming, you may have heard of "procedural programming" as a way of describing code which only consists of procedures (aka, functions) calling other functions, without any higher abstractions. You may have been taught that classes were the *proper* way to transform procedural-style "spaghetti code" into well-formed, well-organized code.

Of course, if you have experience with "functional programming" (Monads, etc.), you know very well that classes are just one of several common design patterns. But for others, this may be the first time you've asked yourself if classes really are a fundamental foundation for code, or if they are an optional abstraction on top of code.

Some languages (like Java) don't give you the choice, so it's not very *optional* at all -- everything's a class. Other languages like C/C++ or PHP give you both procedural and class-oriented syntaxes, and it's left more to the developer's choice which style or mixture of styles is appropriate.

JavaScript "Classes"

Where does JavaScript fall in this regard? JS has had *some* class-like syntactic elements (like `new` and `instanceof`) for quite awhile, and more recently in ES6, some additions, like the `class` keyword (see Appendix A).

But does that mean JavaScript actually *has* classes? Plain and simple: No.

Since classes are a design pattern, you *can*, with quite a bit of effort (as we'll see throughout the rest of this chapter), implement approximations for much of classical class functionality. JS tries to satisfy the extremely pervasive *desire* to design with classes by providing seemingly class-like syntax.

While we may have a syntax that looks like classes, it's as if JavaScript mechanics are fighting against you using the *class design pattern*, because behind the curtain, the mechanisms that you

build on are operating quite differently. Syntactic sugar and (extremely widely used) JS "Class" libraries go a long way toward hiding this reality from you, but sooner or later you will face the fact that the *classes* you have in other languages are not like the "classes" you're faking in JS.

What this boils down to is that classes are an optional pattern in software design, and you have the choice to use them in JavaScript or not. Since many developers have a strong affinity to class oriented software design, we'll spend the rest of this chapter exploring what it takes to maintain the illusion of classes with what JS provides, and the pain points we experience.

’ Class Mechanics

In many class-oriented languages, the "standard library" provides a "stack" data structure (push, pop, etc.) as a *stack class*. This class would have an internal set of variables that stores the data, and it would have a set of publicly accessible behaviors ("methods") provided by the class, which gives your code the ability to interact with the (hidden) data (adding & removing data, etc.).

But in such languages, you don't really operate directly on *stack* (unless making a **Static** class member reference, which is outside the scope of our discussion). The *stack class* is merely an abstract explanation of what *any* "stack" should do, but it's not itself *a* "stack". You must **instantiate** the *stack class* before you have a concrete data structure *thing* to operate against.

’ Building

The traditional metaphor for "class" and "instance" based thinking comes from a building construction.

An architect plans out all the characteristics of a building: how wide, how tall, how many windows and in what locations, even what type of material to use for the walls and roof. She doesn't necessarily care, at this point, *where* the building will be built, nor does she care *how many* copies of that building will be built.

She also doesn't care very much about the contents of the building -- the furniture, wall paper, ceiling fans, etc. -- only what type of structure they will be contained by.

The architectural blue-prints she produces are only *plans* for a building. They don't actually constitute a building we can walk into and sit down. We need a builder for that task. A builder will take those plans and follow them, exactly, as he *builds* the building. In a very real sense, he is *copying* the intended characteristics from the plans to the physical building.

Once complete, the building is a physical instantiation of the blue-print plans, hopefully an essentially perfect *copy*. And then the builder can move to the open lot next door and do it all over again, creating yet another *copy*.

The relationship between building and blue-print is indirect. You can examine a blue-print to understand how the building was structured, for any parts where direct inspection of the building itself was insufficient. But if you want to open a door, you have to go to the building itself -- the blue-print merely has lines drawn on a page that *represent* where the door should be.

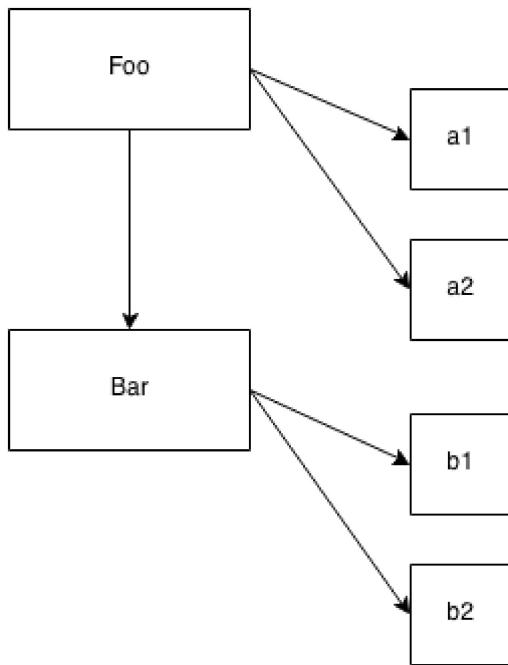
A class is a blue-print. To actually *get* an object we can interact with, we must build (aka, "instantiate") something from the class. The end result of such "construction" is an object, typically

called an "instance", which we can directly call methods on and access any public data properties from, as necessary.

This object is a *copy* of all the characteristics described by the class.

You likely wouldn't expect to walk into a building and find, framed and hanging on the wall, a copy of the blue-prints used to plan the building, though the blue-prints are probably on file with a public records office. Similarly, you don't generally use an object instance to directly access and manipulate its class, but it is usually possible to at least determine *which class* an object instance comes from.

It's more useful to consider the direct relationship of a class to an object instance, rather than any indirect relationship between an object instance and the class it came from. **A class is instantiated into object form by a copy operation.**



As you can see, the arrows move from left to right, and from top to bottom, which indicates the copy operations that occur, both conceptually and physically.

◦ Constructor

Instances of classes are constructed by a special method of the class, usually of the same name as the class, called a *constructor*. This method's explicit job is to initialize any information (state) the instance will need.

For example, consider this loose pseudo-code (invented syntax) for classes:

```

class CoolGuy {
    specialTrick = nothing

    CoolGuy( trick ) {
        specialTrick = trick
    }

    showOff() {
  
```

```

        output( "Here's my trick: ", specialTrick )
    }
}

```

To make a `CoolGuy` instance, we would call the class constructor:

```

Joe = new CoolGuy( "jumping rope" )

Joe.showOff() // Here's my trick: jumping rope

```

Notice that the `CoolGuy` class has a constructor `CoolGuy()`, which is actually what we call when we say `new CoolGuy(...)`. We get an object back (an instance of our class) from the constructor, and we can call the method `showoff()`, which prints out that particular `CoolGuys` special trick.

Obviously, jumping rope makes Joe a pretty cool guy.

The constructor of a class *belongs* to the class, almost universally with the same name as the class. Also, constructors pretty much always need to be called with `new` to let the language engine know you want to construct a *new* class instance.

’ Class Inheritance

In class-oriented languages, not only can you define a class which can be instantiated itself, but you can define another class that **inherits** from the first class.

The second class is often said to be a "child class" whereas the first is the "parent class". These terms obviously come from the metaphor of parents and children, though the metaphors here are a bit stretched, as you'll see shortly.

When a parent has a biological child, the genetic characteristics of the parent are copied into the child. Obviously, in most biological reproduction systems, there are two parents who co-equally contribute genes to the mix. But for the purposes of the metaphor, we'll assume just one parent.

Once the child exists, he or she is separate from the parent. The child was heavily influenced by the inheritance from his or her parent, but is unique and distinct. If a child ends up with red hair, that doesn't mean the parent's hair *was* or automatically *becomes* red.

In a similar way, once a child class is defined, it's separate and distinct from the parent class. The child class contains an initial copy of the behavior from the parent, but can then override any inherited behavior and even define new behavior.

It's important to remember that we're talking about parent and child **classes**, which aren't physical things. This is where the metaphor of parent and child gets a little confusing, because we actually should say that a parent class is like a parent's DNA and a child class is like a child's DNA. We have to make (aka "instantiate") a person out of each set of DNA to actually have a physical person to have a conversation with.

Let's set aside biological parents and children, and look at inheritance through a slightly different lens: different types of vehicles. That's one of the most canonical (and often groan-worthy)

metaphors to understand inheritance.

Let's revisit the vehicle and car discussion from earlier in this chapter. Consider this loose pseudo-code (invented syntax) for inherited classes:

```

class Vehicle {
    engines = 1

    ignition() {
        output( "Turning on my engine." )
    }

    drive() {
        ignition()
        output( "Steering and moving forward!" )
    }
}

class Car inherits Vehicle {
    wheels = 4

    drive() {
        inherited:drive()
        output( "Rolling on all ", wheels, " wheels!" )
    }
}

class SpeedBoat inherits Vehicle {
    engines = 2

    ignition() {
        output( "Turning on my ", engines, " engines." )
    }

    pilot() {
        inherited:drive()
        output( "Speeding through the water with ease!" )
    }
}

```

Note: For clarity and brevity, constructors for these classes have been omitted.

We define the `Vehicle` class to assume an engine, a way to turn on the ignition, and a way to drive around. But you wouldn't ever manufacture just a generic "vehicle", so it's really just an abstract concept at this point.

So then we define two specific kinds of vehicle: `car` and `SpeedBoat`. They each inherit the general characteristics of `Vehicle`, but then they specialize the characteristics appropriately for each kind. A car needs 4 wheels, and a speed boat needs 2 engines, which means it needs extra attention to turn on the ignition of both engines.

Polymorphism

Car defines its own `drive()` method, which overrides the method of the same name it inherited from `Vehicle`. But then, `Cars` `drive()` method calls `inherited:drive()`, which indicates that `Car` can reference the original pre-overridden `drive()` it inherited. `SpeedBoats` `pilot()` method also makes a reference to its inherited copy of `drive()`.

This technique is called "polymorphism", or "virtual polymorphism". More specifically to our current point, we'll call it "relative polymorphism".

Polymorphism is a much broader topic than we will exhaust here, but our current "relative" semantics refers to one particular aspect: the idea that any method can reference another method (of the same or different name) at a higher level of the inheritance hierarchy. We say "relative" because we don't absolutely define which inheritance level (aka, class) we want to access, but rather relatively reference it by essentially saying "look one level up".

In many languages, the keyword `super` is used, in place of this example's `inherited:`, which leans on the idea that a "super class" is the parent/ancestor of the current class.

Another aspect of polymorphism is that a method name can have multiple definitions at different levels of the inheritance chain, and these definitions are automatically selected as appropriate when resolving which methods are being called.

We see two occurrences of that behavior in our example above: `drive()` is defined in both `Vehicle` and `Car`, and `ignition()` is defined in both `Vehicle` and `SpeedBoat`.

Note: Another thing that traditional class-oriented languages give you via `super` is a direct way for the constructor of a child class to reference the constructor of its parent class. This is largely true because with real classes, the constructor belongs to the class. However, in JS, it's the reverse -- it's actually more appropriate to think of the "class" belonging to the constructor (the `Foo.prototype...` type references). Since in JS the relationship between child and parent exists only between the two `.prototype` objects of the respective constructors, the constructors themselves are not directly related, and thus there's no simple way to relatively reference one from the other (see Appendix A for ES6 `class` which "solves" this with `super`).

An interesting implication of polymorphism can be seen specifically with `ignition()`. Inside `pilot()`, a relative-polymorphic reference is made to (the inherited) `Vehicle`'s version of `drive()`. But that `drive()` references an `ignition()` method just by name (no relative reference).

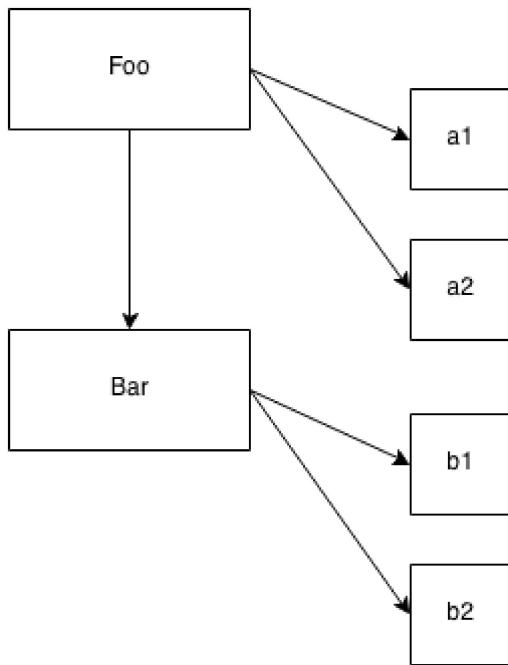
Which version of `ignition()` will the language engine use, the one from `Vehicle` or the one from `SpeedBoat`? **It uses the SpeedBoat version of `ignition()`.** If you were to instantiate `Vehicle` class itself, and then call its `drive()`, the language engine would instead just use `Vehicle`'s `ignition()` method definition.

Put another way, the definition for the method `ignition()` *polymorphs* (changes) depending on which class (level of inheritance) you are referencing an instance of.

This may seem like overly deep academic detail. But understanding these details is necessary to properly contrast similar (but distinct) behaviors in JavaScript's `[[Prototype]]` mechanism.

When classes are inherited, there is a way **for the classes themselves** (not the object instances created from them!) to *relatively* reference the class inherited from, and this relative reference is usually called `super`.

Remember this figure from earlier:



Notice how for both instantiation (`a1`, `a2`, `b1`, and `b2`) *and* inheritance (`Bar`), the arrows indicate a copy operation.

Conceptually, it would seem a child class `Bar` can access behavior in its parent class `Foo` using a relative polymorphic reference (aka, `super`). However, in reality, the child class is merely given a copy of the inherited behavior from its parent class. If the child "overrides" a method it inherits, both the original and overridden versions of the method are actually maintained, so that they are both accessible.

Don't let polymorphism confuse you into thinking a child class is linked to its parent class. A child class instead gets a copy of what it needs from the parent class. **Class inheritance implies copies.**

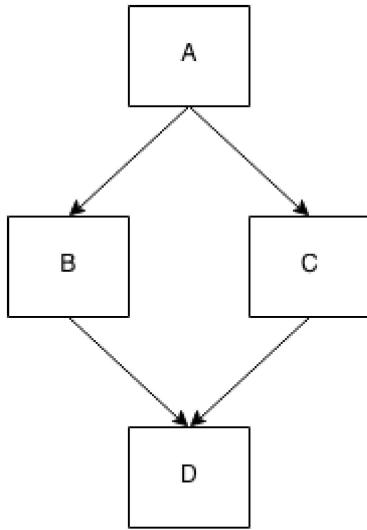
‣ Multiple Inheritance

Recall our earlier discussion of parent(s) and children and DNA? We said that the metaphor was a bit weird because biologically most offspring come from two parents. If a class could inherit from two other classes, it would more closely fit the parent/child metaphor.

Some class-oriented languages allow you to specify more than one "parent" class to "inherit" from. Multiple-inheritance means that each parent class definition is copied into the child class.

On the surface, this seems like a powerful addition to class-orientation, giving us the ability to compose more functionality together. However, there are certainly some complicating questions that arise. If both parent classes provide a method called `drive()`, which version would a `drive()` reference in the child resolve to? Would you always have to manually specify which parent's `drive()` you meant, thus losing some of the gracefulness of polymorphic inheritance?

There's another variation, the so called "Diamond Problem", which refers to the scenario where a child class "D" inherits from two parent classes ("B" and "C"), and each of those in turn inherits from a common "A" parent. If "A" provides a method `drive()`, and both "B" and "C" override (polymorph) that method, when D references `drive()`, which version should it use (`B:drive()` or `C:drive()`)?



These complications go even much deeper than this quick glance. We address them here only so we can contrast to how JavaScript's mechanisms work.

JavaScript is simpler: it does not provide a native mechanism for "multiple inheritance". Many see this is a good thing, because the complexity savings more than make up for the "reduced" functionality. But this doesn't stop developers from trying to fake it in various ways, as we'll see next.

› Mixins

JavaScript's object mechanism does not *automatically* perform copy behavior when you "inherit" or "instantiate". Plainly, there are no "classes" in JavaScript to instantiate, only objects. And objects don't get copied to other objects, they get *linked together* (more on that in Chapter 5).

Since observed class behaviors in other languages imply copies, let's examine how JS developers **fake** the *missing* copy behavior of classes in JavaScript: mixins. We'll look at two types of "mixin": **explicit** and **implicit**.

› Explicit Mixins

Let's again revisit our `Vehicle` and `Car` example from before. Since JavaScript will not automatically copy behavior from `Vehicle` to `Car`, we can instead create a utility that manually copies. Such a utility is often called `extend(..)` by many libraries/frameworks, but we will call it `mixin(..)` here for illustrative purposes.

```

// vastly simplified `mixin(..)` example:
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        // only copy if not already present
        if (!(key in targetObj)) {
            targetObj[key] = sourceObj[key];
        }
    }
    return targetObj;
}
  
```

```

var Vehicle = {
    engines: 1,

    ignition: function() {
        console.log( "Turning on my engine." );
    },

    drive: function() {
        this.ignition();
        console.log( "Steering and moving forward!" );
    }
};

var Car = mixin( Vehicle, {
    wheels: 4,

    drive: function() {
        Vehicle.drive.call( this );
        console.log( "Rolling on all " + this.wheels + " wheels!" );
    }
} );

```

Note: Subtly but importantly, we're not dealing with classes anymore, because there are no classes in JavaScript. `Vehicle` and `Car` are just objects that we make copies from and to, respectively.

`Car` now has a copy of the properties and functions from `Vehicle`. Technically, functions are not actually duplicated, but rather *references* to the functions are copied. So, `Car` now has a property called `ignition`, which is a copied reference to the `ignition()` function, as well as a property called `engines` with the copied value of `1` from `Vehicle`.

`Car` *already* had a `drive` property (function), so that property reference was not overridden (see the `if` statement in `mixin(...)` above).

› "Polymorphism" Revisited

Let's examine this statement: `Vehicle.drive.call(this)`. This is what I call "explicit pseudo-polymorphism". Recall in our previous pseudo-code this line was `inherited:drive()`, which we called "relative polymorphism".

JavaScript does not have (prior to ES6; see Appendix A) a facility for relative polymorphism. So, because both `Car` and `Vehicle` had a function of the same name: `drive()`, to distinguish a call to one or the other, we must make an absolute (not relative) reference. We explicitly specify the `Vehicle` object by name, and call the `drive()` function on it.

But if we said `Vehicle.drive()`, the `this` binding for that function call would be the `Vehicle` object instead of the `Car` object (see Chapter 2), which is not what we want. So, instead we use `.call(this)` (Chapter 2) to ensure that `drive()` is executed in the context of the `Car` object.

Note: If the function name identifier for `Car.drive()` hadn't overlapped with (aka, "shadowed"; see Chapter 5) `Vehicle.drive()`, we wouldn't have been exercising "method polymorphism". So, a reference to `Vehicle.drive()` would have been copied over by the `mixin(...)` call, and we could

have accessed directly with `this.drive()`. The chosen identifier overlap **shadowing** is *why* we have to use the more complex *explicit pseudo-polymorphism* approach.

In class-oriented languages, which have relative polymorphism, the linkage between `Car` and `Vehicle` is established once, at the top of the class definition, which makes for only one place to maintain such relationships.

But because of JavaScript's peculiarities, explicit pseudo-polymorphism (because of shadowing!) creates brittle manual/explicit linkage in **every single function where you need such a (pseudo-)polymorphic reference**. This can significantly increase the maintenance cost. Moreover, while explicit pseudo-polymorphism can emulate the behavior of "multiple inheritance", it only increases the complexity and brittleness.

The result of such approaches is usually more complex, harder-to-read, and harder-to-maintain code. **Explicit pseudo-polymorphism should be avoided wherever possible**, because the cost outweighs the benefit in most respects.

› Mixing Copies

Recall the `mixin(..)` utility from above:

```
// vastly simplified `mixin()` example:
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        // only copy if not already present
        if (!(key in targetObj)) {
            targetObj[key] = sourceObj[key];
        }
    }

    return targetObj;
}
```

Now, let's examine how `mixin(..)` works. It iterates over the properties of `sourceObj` (`Vehicle` in our example) and if there's no matching property of that name in `targetObj` (`Car` in our example), it makes a copy. Since we're making the copy after the initial object exists, we are careful to not copy over a target property.

If we made the copies first, before specifying the `Car` specific contents, we could omit this check against `targetObj`, but that's a little more clunky and less efficient, so it's generally less preferred:

```
// alternate mixin, less "safe" to overwrites
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        targetObj[key] = sourceObj[key];
    }

    return targetObj;
}

var Vehicle = {
```

```
// ...
};

// first, create an empty object with
// Vehicle's stuff copied in
var Car = mixin( Vehicle, { } );

// now copy the intended contents into Car
mixin( {
    wheels: 4,

    drive: function() {
        // ...
    }
}, Car );
```

Either approach, we have explicitly copied the non-overlapping contents of `Vehicle` into `Car`. The name "mixin" comes from an alternate way of explaining the task: `car` has `Vehicle`'s contents **mixed-in**, just like you mix in chocolate chips into your favorite cookie dough.

As a result of the copy operation, `car` will operate somewhat separately from `Vehicle`. If you add a property onto `car`, it will not affect `Vehicle`, and vice versa.

Note: A few minor details have been skimmed over here. There are still some subtle ways the two objects can "affect" each other even after copying, such as if they both share a reference to a common object (such as an array).

Since the two objects also share references to their common functions, that means that **even manual copying of functions (aka, mixins) from one object to another doesn't actually emulate the real duplication from class to instance that occurs in class-oriented languages**.

JavaScript functions can't really be duplicated (in a standard, reliable way), so what you end up with instead is a **duplicated reference** to the same shared function object (functions are objects; see Chapter 3). If you modified one of the shared **function objects** (like `ignition()`) by adding properties on top of it, for instance, both `Vehicle` and `Car` would be "affected" via the shared reference.

Explicit mixins are a fine mechanism in JavaScript. But they appear more powerful than they really are. Not much benefit is *actually* derived from copying a property from one object to another, **as opposed to just defining the properties twice**, once on each object. And that's especially true given the function-object reference nuance we just mentioned.

If you explicitly mix-in two or more objects into your target object, you can **partially emulate** the behavior of "multiple inheritance", but there's no direct way to handle collisions if the same method or property is being copied from more than one source. Some developers/libraries have come up with "late binding" techniques and other exotic work-arounds, but fundamentally these "tricks" are *usually* more effort (and lesser performance!) than the pay-off.

Take care only to use explicit mixins where it actually helps make more readable code, and avoid the pattern if you find it making code that's harder to trace, or if you find it creates unnecessary or unwieldy dependencies between objects.

If it starts to get *harder* to properly use mixins than before you used them, you should probably stop using mixins. In fact, if you have to use a complex library/utility to work out all these details, it might be a sign that you're going about it the harder way, perhaps unnecessarily. In Chapter 6, we'll try to distill a simpler way that accomplishes the desired outcomes without all the fuss.

› Parasitic Inheritance

A variation on this explicit mixin pattern, which is both in some ways explicit and in other ways implicit, is called "parasitic inheritance", popularized mainly by Douglas Crockford.

Here's how it can work:

```
// "Traditional JS Class" `Vehicle`
function Vehicle() {
    this.engines = 1;
}
Vehicle.prototype.ignition = function() {
    console.log( "Turning on my engine." );
};
Vehicle.prototype.drive = function() {
    this.ignition();
    console.log( "Steering and moving forward!" );
};

// "Parasitic Class" `Car`
function Car() {
    // first, `car` is a `Vehicle`
    var car = new Vehicle();

    // now, let's modify our `car` to specialize it
    car.wheels = 4;

    // save a privileged reference to `Vehicle::drive()`
    var vehDrive = car.drive;

    // override `Vehicle::drive()`
    car.drive = function() {
        vehDrive.call( this );
        console.log( "Rolling on all " + this.wheels + " wheels!" );
    };

    return car;
}

var myCar = new Car();

myCar.drive();
// Turning on my engine.
// Steering and moving forward!
// Rolling on all 4 wheels!
```

As you can see, we initially make a copy of the definition from the `Vehicle` "parent class" (object), then mixin our "child class" (object) definition (preserving privileged parent-class references as needed), and pass off this composed object `car` as our child instance.

Note: when we call `new Car()`, a new object is created and referenced by `Cars` this reference (see Chapter 2). But since we don't use that object, and instead return our own `car` object, the initially created object is just discarded. So, `Car()` could be called without the `new` keyword, and the functionality above would be identical, but without the wasted object creation/garbage-collection.

Implicit Mixins

Implicit mixins are closely related to *explicit pseudo-polymorphism* as explained previously. As such, they come with the same caveats and warnings.

Consider this code:

```
var Something = {
    cool: function() {
        this.greeting = "Hello World";
        this.count = this.count ? this.count + 1 : 1;
    }
};

Something.cool();
Something.greeting; // "Hello World"
Something.count; // 1

var Another = {
    cool: function() {
        // implicit mixin of `Something` to `Another`
        Something.cool.call( this );
    }
};

Another.cool();
Another.greeting; // "Hello World"
Another.count; // 1 (not shared state with `Something`)
```

With `Something.cool.call(this)`, which can happen either in a "constructor" call (most common) or in a method call (shown here), we essentially "borrow" the function `Something.cool()` and call it in the context of `Another` (via its `this`binding; see Chapter 2) instead of `Something`. The end result is that the assignments that `Something.cool()` makes are applied against the `Another` object rather than the `Something` object.

So, it is said that we "mixed in" `Something`'s behavior with (or into) `Another`.

While this sort of technique seems to take useful advantage of `this` rebinding functionality, it is the brittle `Something.cool.call(this)` call, which cannot be made into a relative (and thus more flexible) reference, that you should **heed with caution**. Generally, **avoid such constructs where possible** to keep cleaner and more maintainable code.

› Review (TL;DR)

Classes are a design pattern. Many languages provide syntax which enables natural class-oriented software design. JS also has a similar syntax, but it behaves **very differently** from what you're used to with classes in those other languages.

Classes mean copies.

When traditional classes are instantiated, a copy of behavior from class to instance occurs. When classes are inherited, a copy of behavior from parent to child also occurs.

Polymorphism (having different functions at multiple levels of an inheritance chain with the same name) may seem like it implies a referential relative link from child back to parent, but it's still just a result of copy behavior.

JavaScript **does not automatically** create copies (as classes imply) between objects.

The mixin pattern (both explicit and implicit) is often used to *sort of* emulate class copy behavior, but this usually leads to ugly and brittle syntax like explicit pseudo-polymorphism (`OtherObj.methodName.call(this, ...)`), which often results in harder to understand and maintain code.

Explicit mixins are also not exactly the same as class *copy*, since objects (and functions!) only have shared references duplicated, not the objects/functions duplicated themselves. Not paying attention to such nuance is the source of a variety of gotchas.

In general, faking classes in JS often sets more landmines for future coding than solving present *real* problems.

Глава 2: Весь `this` теперь приобретает смысл!

В главе 1 мы отбросили различные ложные представления о `this` и взамен изучили, что привязка `this` происходит при каждом вызове функции, целиком на основании ее **места вызова** (как была вызвана функция).

Точка вызова

Чтобы понять привязку `this`, мы должны понять что такое точка вызова: это место в коде, где была вызвана функция (**не там, где она объявлена**). Мы должны исследовать точку вызова, чтобы ответить на вопрос: на что же *этот this* указывает?

В общем поиск точки вызова выглядит так: "найти откуда вызывается функция", но это не всегда так уж легко, поскольку определенные шаблоны кодирования могут ввести в заблуждение относительно *истинной* точки вызова.

Важно поразмышлять над **стеком вызовов** (стеком функций, которые были вызваны, чтобы привести нас к текущей точке исполнения кода). Точка вызова, которая нас интересует, находится *в вызове перед* текущей выполняемой функцией.

Продемонстрируем стек вызовов и точку вызова:

```
function baz() {
  // стек вызовов: `baz`
  // поэтому наша точка вызова — глобальная область видимости

  console.log( "baz" );
  bar(); // <-- точка вызова для `bar`
}

function bar() {
  // стек вызовов: `baz` -> `bar`
  // поэтому наша точка вызова в `baz`

  console.log( "bar" );
  foo(); // <-- точка вызова для `foo`
}

function foo() {
  // стек вызовов: `baz` -> `bar` -> `foo`
  // поэтому наша точка вызова в `bar`

  console.log( "foo" );
}

baz(); // <-- точка вызова для `baz`
```

Позаботьтесь при анализе кода о том, чтобы найти настоящую точку вызова (из стека вызовов), поскольку это единственная вещь, которая имеет значение для привязки `this`.

Примечание: Вы можете мысленно визуализировать стек вызовов посмотрев цепочку вызовов функций в том порядке, в котором мы это делали в комментариях в коде выше. Но это утомительно и чревато ошибками. Другой путь посмотреть стек вызовов — это использование инструмента отладки в вашем браузере. Во многих современных настольных браузерах есть встроенные инструменты разработчика, включающие JS-отладчик. В вышеприведенном коде вы могли бы поставить точку остановки в такой утилите на первой строке функции `foo()` или просто вставить оператор `debugger;` в первую строку. Как только вы запустите страницу, отладчик остановится в этом месте и покажет вам список функций, которые были вызваны, чтобы добраться до этой строки, каковые и будут являться необходимым стеком вызовов. Таким образом, если вы пытаетесь выяснить привязку `this`, используйте инструменты разработчика для получения стека вызовов, затем найдите второй элемент стека от его вершины и это и будет реальная точка вызова.

› Ничего кроме правил

Теперь обратим наш взор на то, как точка вызова определяет на что будет указывать `this` во время выполнения функции.

Вам нужно изучить точку вызова и определить какое из 4 правил применяется. Сначала разъясним каждое из 4 правил по отдельности, а затем проиллюстрируем их порядок приоритета, для случаев когда к точке вызова могут применяться несколько правил сразу.

› Привязка по умолчанию

Первое правило, которое мы изучим, исходит из самого распространенного случая вызовов функции: отдельный вызов функции. Представьте себе *это* правило `this` как правило, действующее по умолчанию когда остальные правила не применяются.

Рассмотрим такой код:

```
function foo() {  
    console.log( this.a );  
}  
  
var a = 2;  
  
foo(); // 2
```

Первая вещь, которую можно отметить, если вы еще не сделали этого, то, что переменные, объявленные в глобальной области видимости, как например `var a = 2`, являются синонимами глобальных свойств-объектов с таким же именем. Они не являются копиями друг друга, они и есть одно и то же. Представляйте их как две стороны одной монеты.

Во-вторых, видно, что когда вызывается `foo()` `this.a` указывает на нашу глобальную переменную `a`. Почему? Потому что в этом случае, для `this` применяется *привязка по умолчанию* при вызове функции и поэтому `this` указывает на глобальный объект.

Откуда мы знаем, что здесь применяется *привязка по умолчанию*? Мы исследуем точку вызова, чтобы выяснить как вызывается `foo()`. В нашем примере кода `foo()` вызывается по прямой, необернутой ссылке на функцию. Ни одного из демонстрируемых далее правил тут не будет применено, поэтому вместо них применяется *привязка по умолчанию*.

Когда включен `strict mode`, объект '`global`' не подпадает под действие *привязки по умолчанию*, поэтому в противоположность обычному режиму `this` устанавливается в `undefined`.

```
function foo() {
    "use strict";

    console.log( this.a );
}

var a = 2;

foo(); // TypeError: `this` is `undefined`
```

Едва уловимая, но важная деталь: даже если все правила привязки `this` целиком основываются на точке вызова, глобальный объект подпадает под *привязку по умолчанию* только если **содержимое `foo()` не выполняется в режиме `strict mode`**; Состояние `strict mode` в точке вызова `foo()` не имеет значения.

```
function foo() {
    console.log( this.a );
}

var a = 2;

(function(){
    "use strict";

    foo(); // 2
})();
```

Примечание: К намеренному смешиванию включения и выключения `strict mode` в коде обычно относятся неодобрительно. Вся программа пожалуй должна быть либо **строгой**, либо **нестрогой**. Однако, иногда вы подключаете сторонние библиотеки, в которых этот режим **строгости** отличается от вашего, поэтому нужно отнестись с вниманием к таким едва уловимым деталям совместимости.

‣ Неявная привязка

Рассмотрим еще одно правило: есть ли у точки вызова объект контекста, также называемый как владеющий или содержащий объект, хотя эти альтернативные термины могут немного вводить в заблуждение.

Рассмотрим:

```

function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

obj.foo(); // 2

```

Во-первых, отметим способ, которым была объявлена `foo()`, а затем позже добавлена как ссылочное свойство в `obj`. Независимо от того была ли `foo()` изначально объявлена в `obj` или добавлена позднее как ссылка (как в вышеприведенном коде), ни в том, ни в другом случае **функция** на самом деле не "принадлежит" или "содержится" в объекте `obj`.

Однако, точка вызова *использует* контекст `obj`, чтобы **ссылаться** на функцию, поэтому можно сказать, что объект `obj` "владеет" или "содержит" **ссылку на функцию** в момент вызова функции.

Какое название вы бы ни выбрали для этого шаблона, в момент когда вызывается `foo()`, ей предшествует объектная ссылка на `obj`. Когда есть объект контекста для ссылки на функцию, правило *неявной привязки* говорит о том, что именно *этот* объект и следует использовать для привязки `this` к вызову функции.

Поскольку `obj` является `this` для вызова `foo()`, `this.a` — синоним `obj.a`.

Только верхний/последний уровень ссылки на свойство объекта в цепочке имеет значение для точки вызова. Например:

```

function foo() {
    console.log( this.a );
}

var obj2 = {
    a: 42,
    foo: foo
};

var obj1 = {
    a: 2,
    obj2: obj2
};

obj1.obj2.foo(); // 42

```

› Неявно потерянный

Одним из самых распространенных недовольств, которые вызывает привязка `this` — когда *неявно привязанная* функция теряет эту привязку, что обычно означает что она

вернется к привязке по умолчанию, либо объекта `global`, либо `undefined`, в зависимости от режима `strict mode`.

Представим такой код:

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var bar = obj.foo; // ссылка/алиас на функцию!

var a = "ой, глобальная"; // `a` также и свойство глобального объекта

bar(); // "ой, глобальная"
```

Несмотря на то, что `bar` по всей видимости ссылка на `obj.foo`, фактически, это на самом деле другая ссылка на саму `foo`. Более того, именно точка вызова тут имеет значение, а точкой вызова является `bar()`, который является прямым непривязанным вызовом, а следовательно применяется *привязка по умолчанию*.

Более неочевидный, более распространенный и более неожиданный путь получить такую ситуацию когда мы предполагаем передать функцию обратного вызова:

```
function foo() {
    console.log( this.a );
}

function doFoo(fn) {
    // `fn` — просто еще одна ссылка на `foo`

    fn(); // <-- точка вызова!
}

var obj = {
    a: 2,
    foo: foo
};

var a = "ой, глобальная"; // `a` еще и переменная в глобальном объекте

doFoo( obj.foo ); // "ой, глобальная"
```

Передаваемый параметр — всего лишь неявное присваивание, а поскольку мы передаем функцию, это неявное присваивание ссылки, поэтому окончательный результат будет таким же как в предыдущем случае.

Что если функция, которую вы передаете в качестве функции обратного вызова, не ваша собственная, а встроенная в язык? Никакой разницы, такой же результат.

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var a = "ой, глобальная"; // `a` еще и переменная в глобальном объекте

setTimeout( obj.foo, 100 ); // "ой, глобальная"
```

Поразмышляйте над этой грубой теоретической псевдо-реализацией `setTimeout()`, которая есть в качестве встроенной в JavaScript-среде:

```
function setTimeout(fn,delay) {
    // подождать (так или иначе) `delay` миллисекунд
    fn(); // <-- точка вызова!
}
```

Достаточно распространенная ситуация, когда функции обратного вызова *теряют* свою привязку `this`, как мы только что видели. Но еще один способ, которым `this` может удивить нас, когда функция, которой мы передаем нашу функцию обратного вызова, намеренно меняет `this` для этого вызова. Обработчики событий в популярных JavaScript-библиотеках часто любят, чтобы в вашей функции обратного вызова `this` принудительно указывал, например, на DOM-элемент, который вызвал это событие. Несмотря на то, что иногда это бывает полезно, в другое время это может прямо таки выводить из себя. К сожалению, эти инструменты редко дают возможность выбирать.

Каким бы путем ни менялся неожиданно `this`, у вас в действительности нет контроля над тем как будет вызвана ваша функция обратного вызова, таким образом у вас нет возможности контролировать точку вызова, чтобы получить заданную привязку. Мы кратко рассмотрим способ "починки" этой проблемы *починив* `this`.

Явная привязка

В случае *неявной привязки*, как мы только что видели, нам требуется менять объект, о котором идет речь, чтобы включить в него функцию и использовать эту ссылку на свойство-функцию, чтобы опосредованно (неявно) привязать `this` к этому объекту.

Но, что если вам надо явно использовать при вызове функции указанный объект для привязки `this`, без помещения ссылки на свойство-функцию в объект?

У "всех" функций в языке есть несколько инструментов, доступных для них (через их `[[Прототип]]`, о котором подробности будут позже), которые могут оказаться полезными в

решении этой задачи. Говоря конкретнее, у функций есть методы `call(..)` и `apply(..)`. Технически, управляющие среды JavaScript иногда обеспечивают функции, которые настолько специфичны, что у них нет такой функциональности. Но таких мало. Абсолютное большинство предоставляемых функций и конечно все функции, которые создете вы сами, безусловно имеют доступ к `call(..)` и `apply(..)`.

Как работают эти инструменты? Они оба принимают в качестве первого параметра объект, который будет использоваться в качестве `this`, а затем вызывают функцию с указанным `this`. Поскольку вы явно указываете какой `this` вы хотите использовать, мы называем такой способ *явной привязкой*.

Представим такой код:

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2
};

foo.call( obj ); // 2
```

Вызов `foo` с *явной привязкой* посредством `foo.call(..)` позволяет нам указать, что `this` будет `obj`.

Если в качестве привязки `this` вы передадите примитивное значение (типа `string`, `boolean` или `number`), то это примитивное значение будет обернуто в свою объектную форму (`new String(..)`, `new Boolean(..)` или `new Number(..)` соответственно). Часто это называют "упаковка".

**Примечание: ** В отношении привязки `this` `call(..)` и `apply(..)` идентичны. Они по-разному ведут себя с дополнительными параметрами, но мы не будем сейчас на этом останавливаться.

К сожалению, *явная привязка* сама по себе все-таки не предлагает никакого решения для указанной ранее проблемы "потери" функцией ее привязки `this`, либо оставляет это на усмотрение фреймворка.

› Жесткая привязка

Но поиграв с вариациями на тему *явной привязки* на самом деле можно получить желаемое. Пример:

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2
};
```

```

var bar = function() {
    foo.call( obj );
};

bar(); // 2
setTimeout( bar, 100 ); // 2

// `bar` жестко привязывает `this` в `foo` к `obj`
// поэтому его нельзя перекрыть
bar.call( window ); // 2

```

Давайте изучим как работает этот вариант. Мы создаем функцию `bar()`, которая внутри вручную вызывает `foo.call(obj)`, таким образом принудительно вызывая `foo` с привязкой `obj` для `this`. Неважно как вы потом вызовете функцию `bar`, она всегда будет вручную вызывать `foo` с `obj`. Такая привязка одновременно явная и сильная, поэтому мы называем ее *жесткой привязкой*.

Самый типичный способ обернуть функцию с *жесткой привязкой* — создать сквозную обертку, передающую все параметры и возвращающую полученное значение:

```

function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = function() {
    return foo.apply( obj, arguments );
};

var b = bar( 3 ); // 2 3
console.log( b ); // 5

```

Еще один способ выразить этот шаблон — создать переиспользуемую вспомогательную функцию:

```

function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

// простая вспомогательная функция `bind`
function bind(fn, obj) {
    return function() {
        return fn.apply( obj, arguments );
    };
}

```

```
var obj = {
  a: 2
};

var bar = bind( foo, obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

Поскольку *жесткая привязка* — очень распространенный шаблон, он есть как встроенный инструмент в ES5: `Function.prototype.bind`, а используется вот так:

```
function foo(something) {
  console.log( this.a, something );
  return this.a + something;
}

var obj = {
  a: 2
};

var bar = foo.bind( obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

`bind(..)` возвращает новую функцию, в которой жестко задан вызов оригинальной функции с именно тем контекстом `this`, который вы указываете.

Примечание: Начиная с ES6, в функции жесткой привязки, выдаваемой `bind(..)`, есть свойство `.name`, наследуемое от исходной функции. Например: у `bar = foo.bind(..)` должно быть в `bar.name` значение "bound foo", которое является названием вызова функции, которое должно отражаться в стеке вызовов.

› "Контексты" в вызовах API

Функции многих библиотек, и разумеется многие встроенные в язык JavaScript и во внешнее окружение функции, предоставляют необязательный параметр, обычно называемый "контекст", который спроектирован как обходной вариант для вас, чтобы не пользоваться `bind(..)`, чтобы гарантировать, что ваша функция обратного вызова использует данный `this`.

Например:

```
function foo(el) {
  console.log( el, this.id );
}

var obj = {
  id: "awesome"
```

```
};
```

```
// используем `obj` как `this` для вызовов `foo(..)`  
[1, 2, 3].forEach( foo, obj ); // 1 awesome 2 awesome 3 awesome
```

Внутренне эти различные функции почти наверняка используют явную привязку через `call(..)` или `apply(..)`, избавляя вас от хлопот.

› Привязка new

Четвертое и последнее правило привяжи `this` потребует от нас переосмысления самого распространенного заблуждения о функциях и объектах в JavaScript.

В традиционных классо-ориентированных языках, "конструкторы" — это особые методы, связанные с классами, таким образом, что когда создается экземпляр класса с помощью операции `new`, вызывается конструктор этого класса. Обычно это выглядит как-то так:

```
something = new MyClass(..);
```

В JavaScript есть операция `new` и шаблон кода, который используется для этого, выглядит в основном идентично такой же операции в классо-ориентированных языках; многие разработчики полагают, что механизм JavaScript выполняет что-то похожее. Однако, на самом деле *нет никакой связи* с классо-ориентированной функциональностью у той, что предполагает использование `new` в JS.

Во-первых, давайте еще раз посмотрим что такое "конструктор" в JavaScript. В JS конструкторы — это **всего лишь функции**, которые, так уж получилось, были вызваны с операцией `new` перед ними. Они ни связаны с классами, ни создают экземпляров классов. Они — даже не особые типы функций. Они — всего лишь обычные функции, которые, по своей сути, "украдены" операцией `new` при их вызове.

Например, функция `Number(..)` действует как конструктор, цитируя спецификацию ES5.1:

15.7.2 Конструктор `Number`

Когда `Number` вызывается как часть выражения `new`, оно является конструктором: оно инициализирует только что созданный объект.

Так что, практически любая старенькая функция, включая встроенные объектные функции, такие как `Number(..)` (см. главу 3), могут вызываться с `new` перед ними и это превратит такой вызов функции в **вызов конструктора**. Это важное, но едва уловимое различие: нет такой вещи как "функции-конструкторы", а скорее есть вызовы, конструирующие из функций.

Когда функция вызывается с указанием перед ней `new`, также известный как вызов конструктора, автоматически выполняются следующие вещи:

1. Создается новенький объект (т.е. конструируется) прямо из воздуха
2. Только что сконструированный объект связывается с [[Прототипом]]
3. Только что сконструированный объект устанавливается как привязка `this` для этого вызова функции

4. За исключением тех случаев, когда функция возвращает свой собственный альтернативный **объект**, вызов функции с new автоматически вернет только что сконструированный объект.

Пункты 1, 3 и 4 применимы к нашему текущему обсуждению. Сейчас мы пропустим пункт 2 и вернемся к нему в главе 5.

Взглянем на такой код:

```
function foo(a) {
    this.a = a;
}

var bar = new foo( 2 );
console.log( bar.a ); // 2
```

Вызывая foo(..) с new впереди нее, мы конструируем новый объект и устанавливаем этот новый объект как this для вызова foo(..). Таким образом new — единственный путь, которым this при вызове функции может быть привязан. Мы называем это *привязкой new*.

Всё по порядку

Итак, теперь мы раскрыли 4 правила привязки this в вызовах функций. Всё, что вам нужно сделать — это найти точку вызова и исследовать ее, чтобы понять какое правило применяется. Но что если к точке вызова можно применить несколько соответствующих правил? Должен быть порядок очередности применения этих правил, а потому далее мы покажем в каком порядке применяются эти правила.

Думаю, совершенно ясно, что *привязка по умолчанию* имеет самый низкий приоритет из четырех. Поэтому мы отложим ее в сторону.

Что должно идти раньше: *неявная привязка* или *явная привязка*? Давайте проверим:

```
function foo() {
    console.log( this.a );
}

var obj1 = {
    a: 2,
    foo: foo
};

var obj2 = {
    a: 3,
    foo: foo
};

obj1.foo(); // 2
obj2.foo(); // 3
```

```
obj1.foo.call( obj2 ); // 3
obj2.foo.call( obj1 ); // 2
```

Итак, явная привязка имеет приоритет над неявной привязкой, что означает, что вы должны спросить себя применима ли сначала явная привязка до проверки на неявную привязку.

Теперь, нам нужно всего лишь указать куда подходит по приоритету привязка new.

```
function foo(something) {
    this.a = something;
}

var obj1 = {
    foo: foo
};

var obj2 = {};

obj1.foo( 2 );
console.log( obj1.a ); // 2

obj1.foo.call( obj2, 3 );
console.log( obj2.a ); // 3

var bar = new obj1.foo( 4 );
console.log( obj1.a ); // 2
console.log( bar.a ); // 4
```

Хорошо, привязка new более приоритетна, чем неявная привязка. Но как вы думаете: привязка new более или менее приоритетна, чем явная привязка?

Примечание: new и call/apply не могут использоваться вместе, поэтому new foo.call(obj1) не корректно, чтобы сравнить напрямую привязку new с явной привязкой. Но мы все-таки можем использовать жесткую привязку, чтобы проверить приоритет этих двух правил.

До того, как мы начнем исследовать всё это на примере кода, постарайтесь вспомнить как физически работает жесткая привязка, которая есть в Function.prototype.bind(..), которая создает новую функцию-обертку, и в ней жестко задано игнорировать ее собственную привязку this (какой бы она ни была) и использовать указанную вручную нами.

По этой причине, кажется очевидным предполагать, что жесткая привязка (которая является формой явной привязки) более приоритетна, чем привязка new, а потому и не может быть перекрыта действием new.

Давайте проверим:

```
function foo(something) {
    this.a = something;
}

var obj1 = {};
```

```
var bar = foo.bind( obj1 );
bar( 2 );
console.log( obj1.a ); // 2

var baz = new bar( 3 );
console.log( obj1.a ); // 2
console.log( baz.a ); // 3
```

Ого! `bar` жестко связан с `obj1`, но `new bar(3)` не меняет `obj1.a` на значение 3 что было бы ожидаемо нами. Вместо этого жестко связанный (с `obj1`) вызов `bar(..)` **может** быть перекрыт с `new`. Поскольку был применен `new`, обратно мы получили новый созданный объект, который мы назвали `baz`, и в результате видно, что в `baz.a` значение 3.

Это должно быть удивительно с учетом ранее рассмотренной "фальшивой" вспомогательной функции привязки:

```
function bind(fn, obj) {
    return function() {
        fn.apply( obj, arguments );
    };
}
```

Если вы порассуждаете о том, как работает код этой вспомогательной функции, в нем нет способа для перекрытия жесткой привязки к `obj` операцией `new` как мы только что выяснили.

Но встроенная `Function.prototype.bind(..)` из ES5 — более сложная, даже очень на самом деле. Вот (немного отформатированный) полифиллинг кода, предоставленный со страницы MDN для функции `bind(..)`:

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function(oThis) {
        if (typeof this !== "function") {
            // наиболее подходящая вещь в ECMAScript 5
            // внутренняя функция IsCallable
            throw new TypeError( "Function.prototype.bind - what " +
                "is trying to be bound is not callable"
            );
        }

        var aArgs = Array.prototype.slice.call( arguments, 1 ),
            fToBind = this,
            fNOP = function(){},
            fBound = function(){
                return fToBind.apply(
                    (
                        this instanceof fNOP &&
                        oThis ? this : oThis
                    ),
                    aArgs.concat( Array.prototype.slice.call( arguments )
                );
            }
    }
}
```

```

        }

;

fNOP.prototype = this.prototype;
fBound.prototype = new fNOP();

return fBound;
};

}

```

Примечание: Полифиллинг `bind(..)`, показанный выше, отличается от встроенной `bind(..)` в ES5, учитывающей функции жесткой привязки, которые используются с `new` (см. ниже почему это может быть полезно). Поскольку полифиллинг не может создавать функцию без `.prototype` так, как это делает встроенная утилита, есть едва уловимый окольный путь, чтобы приблизиться к такому же поведению. Двигайтесь осторожно, если планируете использовать `new` вместе с функцией жесткой привязки и полагаетесь на этот полифиллинг.

Часть, которая позволяет перекрыть `new`:

```

this instanceof fNOP &&
oThis ? this : oThis

// ... and:

fNOP.prototype = this.prototype;
fBound.prototype = new fNOP();

```

Мы не будем на самом деле углубляться в объяснения того, как работает эта хитрость (это сложно и выходит за рамки нашего обсуждения), но по сути утилита определяет была ли вызвана или нет функция жесткой привязки с `new` (в результате получая новый сконструированный объект в качестве ее `this`), и если так, то она использует *этот свежесозданный this* вместо ранее указанной жесткой привязки для `this`.

Почему перекрытие операцией `new` жесткой привязки может быть полезным?

Основная причина такого поведения — чтобы создать функцию (которую можно использовать вместе с `new` для конструирования объектов), которая фактически игнорирует жесткую привязку `this`, но которая инициализирует некоторые или все аргументы функции. Одной из возможностей `bind(..)` является умение сделать аргументы, переданные после первого аргумента, привязки `this`, стандартными аргументами по умолчанию для предшествующей функции (технически называемое "частичным применением", которое является подмножеством "карринга").

Пример:

```

function foo(p1,p2) {
    this.val = p1 + p2;
}

```

```
// используем здесь `null`, т.к. нам нет дела до
// жесткой привязки `this` в этом сценарии, и она
// будет переопределена вызовом с операцией `new` в любом случае!
var bar = foo.bind( null, "p1" );

var baz = new bar( "p2" );

baz.val; // p1p2
```

◦ Определяем this

Теперь можно кратко сформулировать правила для определения `this` по точке вызова функции, в порядке их приоритета. Зададим вопросы в том же порядке и остановимся как только будет применено первое же правило.

1. Функция вызвана с `new` (**привязка new**)? Раз так, то `this` — новый сконструированный объект.

```
var bar = new foo()
```

2. Функция вызвана с `call` или `apply` (**явная привязка**), даже скрыто внутри жесткой привязки в `bind`? Раз так, `this` — явно указанный объект.

```
var bar = foo.call( obj2 )
```

3. Функция вызвана с контекстом (**неявная привязка**), иначе называемым как владеющий или содержащий объект? Раз так, `this` является *тем самым* объектом контекста.

```
var bar = obj1.foo()
```

4. В противном случае, будет `this` по умолчанию (**привязка по умолчанию**). В режиме `strict mode`, это будет `undefined`, иначе будет объект `global`.

```
var bar = foo()
```

Вот и всё. Вот всё, что нужно, чтобы понимать правила привязки `this` для обычных вызовов функций. Ну... почти.

◦ Исключения привязок

Как обычно, из "правил" есть несколько *исключений*.

Поведение привязки `this` в некоторых сценариях может быть неожиданным, там где вы подразумеваете одну привязку, а получаете в итоге поведение привязки по правилу *привязки по умолчанию* (см. ранее).

◦ Проигнорированный this

Если вы передаете `null` или `undefined` в качестве параметра привязки `this` в `call`, `apply` или `bind`, то эти значения фактически игнорируются, а взамен к вызову применяется правило *привязки по умолчанию*.

```
function foo() {
    console.log( this.a );
}

var a = 2;

foo.call( null ); // 2
```

Зачем вам бы понадобилось намеренно передавать что-то подобное `null` в качестве привязки `this`?

Довольно распространено использовать `apply(..)` для распаковки массива значений в качестве параметров вызова функции. Аналогично и `bind(..)` может каррировать параметры (предварительно заданные значения), что может быть очень полезно.

```
function foo(a,b) {
    console.log( "a:" + a + ", b:" + b );
}

// распакуем массив как параметры
foo.apply( null, [2, 3] ); // a:2, b:3

// каррируем с помощью `bind(..)`
var bar = foo.bind( null, 2 );
bar( 3 ); // a:2, b:3
```

Обе этих инструментов требуют указания привязки `this` в качестве первого параметра. Если рассматриваемым функциям не важен `this`, то вам нужно -значение-заменитель, и `null` — это похоже разумный выбор, как мы видели выше.

Примечание: В этой книге мы не уделим этому внимания, но в ES6 есть операция расширения `...`, которая дает возможности синтаксически "развернуть" массив как параметры без необходимости использования `apply(..)`, например как в `foo(...[1,2])`, что равносильно `foo(1,2)` — синтаксически избегая привязки `this`, раз она не нужна. К сожалению, в ES6 нет синтаксической замены каррингу, поэтому параметр `this` вызова `bind(..)` все еще требует внимания.

Однако, есть некоторая скрытая "опасность" в том, чтобы всегда использовать `null`, когда вам не нужна привязка `this`. Если вы когда-нибудь воспользуетесь этим при вызове функции (например, функции сторонней библиотеки, которой вы не управляете) и эта функция *всегда* воспользуется ссылкой на `this`, сработает правило *привязки по умолчанию*, что повлечет за собой ненамеренно ссылку (или еще хуже, мутацию!) на объект `global` (`window` в браузере).

Очевидно, что такая ловушка может привести к ряду очень *трудно диагностируемых/отслеживаемых ошибок*.

› Более безопасный `this`

Пожалуй в некоторой степени "более безопасная" практика — передавать особым образом настроенный объект для `this`, который гарантирует отсутствие побочных эффектов в вашей программе. Заимствуя терминологию из сетевых (и военных) технологий, мы можем создать объект "DMZ" (демилитаризованной зоны (de-militarized zone)) — не более чем полностью пустой, неделегированный (см. главы 5 и 6) объект.

Если всегда передавать DMZ-объект для привязок `this`, которые не требуются, то мы можем быть уверены в том, что любое скрытое/неожидаемое использование `this` будет ограничено пустым объектом, который защитит объект `global` нашей программы от побочных эффектов.

Поскольку этот объект совершенно пустой, лично я люблю давать его переменной имени `φ` (математический символ пустого множества в нижнем регистре). На многих клавиатурах (как например US-раскладка на Mac), этот символ легко можно ввести с помощью `⌥+o` (option+o). В некоторых системах есть возможность назначать горячие клавиши на определенные символы. Если вам не нравится символ `φ` или на вашей клавиатуре сложно набрать такой символ, вы конечно же можете назвать переменную как вам угодно.

Как бы вы ни назвали ее, самый простой путь получить **абсолютно пустой** объект — это `Object.create(null)` (см. главу 5). `Object.create(null)` — похож на `{ }`, но без передачи `Object.prototype`, поэтому он "более пустой", чем просто `{ }`.

```
function foo(a,b) {
    console.log( "a:" + a + ", b:" + b );
}

// наш пустой DMZ-объект
var φ = Object.create( null );

// распаковываем массив как параметры
foo.apply( φ, [2, 3] ); // a:2, b:3

// каррируем с помощью `bind(..)`
var bar = foo.bind( φ, 2 );
bar( 3 ); // a:2, b:3
```

Не только функционально "безопаснее", но еще и стилистически выгоднее использовать `φ`, что семантически отражает желание "Я хочу, чтобы `this` был пустым" немного точнее, чем `null`. Но опять таки, называйте свой DMZ-объект как хотите.

› Косвенность

Еще одной вещью, которую нужно опасаться, является создание (намеренно или нет) "косвенных ссылок" на функции, и в этих случаях, когда такая ссылка на функцию вызывается, то также применяется правило *привязки по умолчанию*.

Самый распространенный путь появления косвенных ссылок — при присваивании:

```
function foo() {
    console.log( this.a );
}
```

```
var a = 2;
var o = { a: 3, foo: foo };
var p = { a: 4 };

o.foo(); // 3
(p.foo = o.foo)(); // 2
```

Результатом выражения присваивания `p.foo = o.foo` будет всего лишь ссылка на внутренний объект функции. В силу этого, настоящая точка вызова - это просто `foo()`, а не `p.foo()` или `o.foo()` как вы могли бы предположить. Согласно вышеприведенным правилам будет применено правило *привязки по умолчанию*.

Напоминание: независимо от того как вы добрались до вызова функции используя правило *привязки по умолчанию*, статус **содержимого** вызванной функции в режиме strict mode, использующего ссылку на `this`, а не точка вызова функции, определяет значение *привязки по умолчанию*: либо объект `global` если не в strict mode ИЛИ `undefined` в strict mode.

Смягчение привязки

Ранее мы отметили, что *жесткая привязка* была одной из стратегий для предотвращения случайного действия правила *привязки по умолчанию* при вызове функции, заставив ее привязаться к указанному `this` (до тех пор, пока вы не используете `new`, чтобы переопределить это поведение!). Проблема в том, что *жесткая привязка* значительно уменьшает гибкость функции, не давая указывать `this` вручную, чтобы перекрыть *неявную привязку* или даже последующие попытки явной привязки.

Было бы неплохо, если бы был путь указать другое умолчание для *привязки по умолчанию* (не `global` или `undefined`), но при этом оставив возможность для функции вручную привязать `this` через технику *неявной* или *явной* привязки.

Можно собрать инструмент так называемой *мягкой привязки*, который эмулирует желаемое поведение.

```
if (!Function.prototype.softBind) {
    Function.prototype.softBind = function(obj) {
        var fn = this,
            curried = [].slice.call( arguments, 1 ),
            bound = function bound() {
                return fn.apply(
                    (!this ||
                        (typeof window !== "undefined" ||
                            this === window) ||
                        (typeof global !== "undefined" &&
                            this === global)
                    ) ? obj : this,
                    curried.concat.apply( curried, arguments )
                );
            };
        bound.prototype = Object.create( fn.prototype );
    };
}
```

```

    return bound;
}
}

```

Инструмент `softBind(...)`, представленный здесь, работает подобно встроенному в ES5 инструменту `bind(...)`, за исключением нашего поведения *мягкой привязки*. Он делает обертку указанной функции с логикой, которая проверяет `this` в момент вызова и если это `global` или `undefined`, использует указанное заданее альтернативное умолчание (`obj`). В противном случае `this` остается как есть. Также этот инструмент дает возможность опционального карринга (см. ранее обсуждение `bind(...)`).

Продемонстрируем его в действии:

```

function foo() {
  console.log("name: " + this.name);
}

var obj = { name: "obj" },
  obj2 = { name: "obj2" },
  obj3 = { name: "obj3" };

var fooOBJ = foo.softBind( obj );

fooOBJ(); // name: obj

obj2.foo = foo.softBind(obj);
obj2.foo(); // name: obj2  <---- смотрите!!!

fooOBJ.call( obj3 ); // name: obj3  <---- смотрите!

setTimeout( obj2.foo, 10 ); // name: obj  <---- возврат к мягкой привяке

```

Для мягкопривязанной версии функции `foo()` можно вручную привязать `this` к `obj2` или `obj3` как показано выше, но он возвращается к `obj` в случае применения *привязки по умолчанию*.

Лексический `this`

В обычных функциях строго соблюдаются 4 правила, которые мы только что рассмотрели. Но в ES6 представлен особый вид функции, которая не использует эти правила: стрелочная функция.

Стрелочные функции обозначаются не ключевым словом `function`, а операцией `=>`, так называемой "жирной стрелкой". Вместо использования четырех стандартных `this`-правил, стрелочные функции заимствуют привязку `this` из окружающей (функции или глобальной) области видимости.

Проиллюстрируем лексическую область видимости стрелочной функции:

```

function foo() {
    // возвращаем стрелочную функцию
    return (a) => {
        // Здесь `this` лексически заимствован из `foo()`
        console.log( this.a );
    };
}

var obj1 = {
    a: 2
};

var obj2 = {
    a: 3
};

var bar = foo.call( obj1 );
bar.call( obj2 ); // 2, а не 3!

```

Стрелочная функция, созданная в `foo()`, лексически захватывает любой `this` в `foo()` во время ее вызова. Поскольку в `foo()` `this` был привязан к `obj1`, `bar` (ссылка на возвращаемую стрелочную функцию) также будет с привязкой `this` к `obj1`. Лексическая привязка стрелочной функции не может быть перекрыта (даже с помощью `new!`).

Самый распространенный вариант использования стрелочной функции — обычно при использовании функций обратного вызова, таких как обработчики событий или таймеры:

```

function foo() {
    setTimeout(() => {
        // Здесь `this` лексически заимствован из `foo()`
        console.log( this.a );
    }, 100);
}

var obj = {
    a: 2
};

foo.call( obj ); // 2

```

Несмотря на то, что стрелочные функции предоставляют альтернативу применению `bind(..)` к функции, чтобы гарантировать определенный `this`, что может выглядеть весьма привлекательно, важно отметить, что они фактически запрещают традиционный механизм `this` в пользу более понятной лексической области видимости. До ES6, у нас уже был довольно распространенный шаблон для выполнения такой задачи, который по сути почти неотличим от сущности стрелочных функций ES6:

```

function foo() {
    var self = this; // лексический захват `this`
    setTimeout( function(){

```

```

        console.log( self.a );
    }, 100 );
}

var obj = {
    a: 2
};

foo.call( obj ); // 2

```

В том время как `self = this` и стрелочные функции обе кажутся хорошим "решением" при нежелании использовать `bind(..)`, они фактически убегают от `this` вместо того, чтобы понять и научиться использовать его.

Если вы застали себя пишущим код в стиле `this`, но большую часть или всё время вы сводите на нет механизм `this` с помощью трюков лексической конструкции `self = this` или стрелочной функции, возможно вам следует сделать что-то одно из этого:

1. Использовать только лексическую область видимости и забыть о фальшивости кода в стиле `this`.
2. Полностью научиться использовать механизмы `this`-стиля, включая применение `bind(..)`, где необходимо, и попытаться избегать трюков "лексического `this`" с помощью `self = this` и стрелочной функции.

Программа может эффективно использовать оба стиля кодирования (лексический и `this`), но внутри одной и той же функции и, разумеется, при одних и тех же видах поисков переменных, смешивание двух этих механизмов обычно приводит к менее обслуживаемому коду, и возможно будет слишком перегруженным, чтобы выглядеть умным.

› Обзор

Определение привязки `this` для вызова функции требует поиска непосредственной точки вызова этой функции. Как уже выяснилось, к точке вызова могут быть применены четыре правила, в *именно таком* порядке приоритета:

1. Вызвана с `new`? Используем только что созданный объект.
2. Вызвана с помощью `call` или `apply` (или `bind`)? Используем указанный объект.
3. Вызвана с объектом контекста, владеющего вызовом функции? Используем этот объект контекста.
4. По умолчанию: `undefined` в режиме `strict mode`, в противном случае объект `global`.

Остерегайтесь случайного/неумышленного вызова с применением правила *привязки по умолчанию*. В случаях, когда вам нужно "безопасно" игнорировать привязку `this`, "DMZ"-объект, подобный `φ = Object.create(null)`, — хорошая замена, защищающая объект `global` от непредусмотренных побочных эффектов.

Вместо четырех стандартных правил привязки стрелочные функции ES6 используют лексическую область видимости для привязки `this`, что означает, что они заимствуют привязку `this` (какой бы она ни была) от вызова своей окружающей функции. Они по существу являются синтаксической заменой `self = this` в до-ES6 коде.

Глава 3: Объекты

В первой и второй главе мы объяснили как `this` указывает на разные объекты, в зависимости от места вызова функции. Но что представляют собой объекты на самом деле и почему нам нужно указывать на них? Мы подробно рассмотрим объекты в этой главе.

Синтаксис

Объекты создаются двумя способами: декларативно (литерально) и с помощью конструктора.

Литеральный синтаксис для объекта выглядит так:

```
var myObj = {  
    key: value  
    // ...  
};
```

Конструкторная форма выглядит так:

```
var myObj = new Object();  
myObj.key = value;
```

Конструкторная и литеральная формы в результате дают одинаковые объекты. Единственное отличие в том, что в литературной форме вы можете добавлять сразу несколько пар ключ-значение, в то время как с конструктором вам нужно добавлять свойства по одному.

Примечание: Конструкторная форма для создания объектов, показанная выше, используется крайне редко. Почти всегда вы предпочтёте использовать литературную форму. Это справедливо и для большинства встроенных объектов (смотрите ниже).

Тип

Объекты -- это основные элементы из которых построена большая часть JS. В JS есть шесть основных типов (в спецификации называются «языковые типы»):

- `string`
- `number`
- `boolean`
- `null`
- `undefined`
- `object`

Обратите внимание, что *простые примитивы* (`string`, `number`, `boolean`, `null`, and `undefined`) сами по себе **не являются** объектами. `null` иногда упоминается как объект, но это заблуждение

произошло из ошибки в языке, которая приводит к тому, что `typeof null` ошибочно возвращает "object". По факту, `null` это самостоятельный примитивный тип.

Есть распространённое заблуждение, что «в JS всё является объектом». Это совсем не так.

Однако, есть несколько специальных подтипов, которые мы можем называть *сложными примитивами*.

`function` -- это подтип объекта (технически, «вызываемый объект»). Говорят, что функции в JS это объекты «первого класса», поскольку в основном они являются обычными объектами и с ними можно работать как с любым другим объектом.

Массивы -- это тоже форма объекта с расширенным поведением. Содержимое массивов организовано более структурировано, чем у обычных объектов.

› Встроенные Объекты

Существует несколько других подтипов объектов, обычно называемых *встроенными объектами*. Некоторые их названия подразумевают, что они непосредственно относятся к соответствующим простым примитивам. На самом деле их отношения гораздо сложнее. Скоро мы их рассмотрим подробнее.

- `String`
- `Number`
- `Boolean`
- `Object`
- `Function`
- `Array`
- `Date`
- `RegExp`
- `Error`

Эти подтипы выглядят как настоящие типы или даже классы, если вы полагаетесь на сходство с другими языками вроде класса `String` в Java.

Но в JS это на самом деле встроенные функции. Каждая из этих встроенных функций может быть использована как конструктор (ага, вызов функции с оператором `new` --смотрите Главу 2), а результатом будет новый сконструированный объект указанного подтипа. Например:

```
var strPrimitive = "I am a string";
typeof strPrimitive; // "string"
strPrimitive instanceof String; // false
var strObject = new String( "I am a string" );
typeof strObject; // "object"
strObject instanceof String; // true
// проверим подтип объекта
Object.prototype.toString.call( strObject ); // [object String]
```

В следующей главе мы подробно рассмотрим как именно работает `Object.prototype.toString`.... Вкратце: мы можем проверить внутренний подтип, заимствовав базовый стандартный метод `toString()`. Как видите, он показывает, что этот `strObject` -- на самом деле объект, созданный конструктором `String`.

Значение примитива «I am a string» это не объект, а литерал примитива и его значение иммутабельно. Чтобы выполнять над ним операции вроде проверки длины, доступа к содержимому символов и т. д. требуется объект `String`.

К счастью, при необходимости язык автоматически приводит примитив "string" к объекту `String`, а значит вам почти никогда не придется дополнительно создавать форму Объекта. Большая часть сообщества JS **настоятельно рекомендует** по возможности использовать литеральную форму вместо конструкторной формы.

Рассмотрим:

```
var strPrimitive = "I am a string";
console.log( strPrimitive.length );      // 13
console.log( strPrimitive.charAt( 3 ) );    // "m"
```

В обоих случаях мы вызываем свойство или метод строчного примитива и движок автоматически преобразует его к объекту `String`, так что свойство/метод работают.

Такое же преобразование происходит между численным литеральным примитивом 40 и обёрткой объекта `new Number(42)` при использовании методов вроде `42.359.toFixed(2)`. То же самое и с объектом `Boolean` из примитива "boolean".

`null` и `undefined` не имеют формы Объекта, только значения их примитивов. Для примера, значения `Date` могут быть созданы только с помощью их конструкторной формы объекта, так как у них нет соответствующей литеральной формы.

`Object`, `Array`, `Function`, и `RegExp` (регулярные выражения) -- это объекты, не зависимо от того используется литеральная или конструкторная форма. Конструкторная форма в некоторых случаях предлагает больше опций для создания, чем литеральная. Поскольку объекты создаются в любом случае, простая литеральная форма почти универсальна. **Используйте конструкторную форму только если вам нужны дополнительные опции.**

Объекты `Error` редко создаются в коде в явном виде. Обычно они создаются автоматически когда появляются исключения. Они могут быть созданы с помощью конструкторной формы `new Error(..)`, но обычно в этом нет необходимости.

Содержимое

Как мы упоминали ранее, содержимое объекта состоит из значений (любого типа), которые хранятся в специально названных местах, которые мы называем свойствами.

Важно отметить, что когда мы говорим о «содержимом» и подразумеваем, что эти значения хранятся прямо внутри объекта, то это лишь абстракция. Движок хранит значения в зависимости от его реализации и может запросто не хранить их *внутри* какого-нибудь

контейнера объекта. Что *действительно* хранится в контейнере, так это названия свойств, которые работают как указатели (технически, *ссылаются*) туда, где хранятся значения.

Рассмотрим:

```
var myObject = {
  a: 2
};
myObject.a;           // 2
myObject["a"];        // 2
```

Чтобы получить значение по *адресу* `a` в `myObject` мы должны использовать либо оператор `.` либо оператор `[]`. Синтаксис `.a` обычно описывают как доступ к «свойству», а синтаксис `["a"]` называют доступ по «ключу». В реальности они оба обращаются по одному *адресу* и выведут одно и то же значение `2`, так что эти термины взаимозаменяемы. Начиная отсюда мы будем использовать наиболее общий термин «доступ к свойству».

Основное различие между двумя синтаксисами в том, что оператор `.` требует, чтобы после него шло название свойства, совместимое с Идентификатором, в то время как синтаксис `[".."]` может принять в качестве имени свойства любую строку, совместимую с UTF-8/unicode.

Также, поскольку синтаксис `[".."]` использует **значение** строки для указания адреса, можно программно сгенерировать значение этой строки, например, так:

```
var wantA = true;
var myObject = {
  a: 2
};
var idx;
if (wantA) {
  idx = "a";
}
// позже
console.log( myObject[idx] ); // 2
```

В объектах названия свойств **всегда** являются строкой. Если вы используете в качестве свойств любые другие значения кроме `string` (примитив), они будут сконвертированы в строку. Это относится и к числам, которые обычно используют как индексы массивов. Так что будьте осторожны и не путайте использование чисел в объектах и массивах.

```
var myObject = { };
myObject[true] = "foo";
myObject[3] = "bar";
myObject[myObject] = "baz";
myObject["true"];           // "foo"
myObject["3"];             // "bar"
myObject["[object Object]"]; // "baz"
```

› Вычисляемые имена свойств

Описанный выше синтаксис доступа к свойствам вида `myObject[...]` полезен когда необходимо использовать результат выражения в качестве ключа, вроде `myObject[prefix + name]`. Но это не сильно помогает при объявлении объектов через объектно-литеральный синтаксис.

ES6 добавляет *вычисляемые имена свойств*, где можно указать выражение, обрамленное `[]`, в качестве пары ключ-значение при литеральном объявлении объекта:

```
var prefix = "foo";
var myObject = {
  [prefix + "bar"]: "hello",
  [prefix + "baz"]: "world"
};
myObject["foobar"]; // hello
myObject["foobaz"]; // world
```

Наиболее распространённым, вероятно, будет использование *вычислимых имен свойств* для `Symbol` в ES6, которые не будут подробно рассматриваться в этой книге. Вкратце, это новый примитивный тип данных, значение которого непрозрачно и неопределено (технически, это значение `-- string`). Вы будете сильно озадачены, работая с *действительным значением Symbol* (которое теоретически может быть разным в разных движках JS), поэтому вы будете использовать имя Символа, вроде `Symbol.Something` (просто выдуманное имя!):

```
var myObject = {
  [Symbol.Something]: "hello world"
};
```

› Свойство против Метода

Если разговор идет о доступе к свойству объекта, некоторые разработчики любят различать когда запрашиваемое значение является функцией.

Интересно, что спецификация делает такое же различие.

Технически функции никогда не «относились» к объектам, поэтому утверждение, что функция, к которой обратились через объект автоматически становится «методом» похоже на растягивание семантики.

Это *правда*, что в некоторых функциях есть указание на `this` и иногда использование `this` связано с указанием на объект в точке вызова. Но такое использование не делает эту функцию более «методной», чем другую, поскольку `this` привязывается динамически во время вызова в точке вызова и поэтому его отношение к объекту в лучшем случае косвенное.

Каждый раз, когда вы запрашиваете свойство объекта, это **обращение к свойству**, вне зависимости от типа значения, которое вы получаете. Если *вдруг* вы получите в результате обращения к этому свойству функцию, она не превратится волшебным образом в «метод».

Нет ничего особенного в том, что функция выводится при обращении к свойству (кроме возможной неявной привязки `this`, как было описано ранее).

Например:

```
function foo() {
  console.log( "foo" );
}

var someFoo = foo;      // переменная указывает на `foo`
var myObject = {
  someFoo: foo
};

foo;                  // function foo(){..}
someFoo;              // function foo(){..}
myObject.someFoo;     // function foo(){..}
```

`someFoo` и `myObject.someFoo` это лишь две разных ссылки на одну функцию и ни одна из них не подразумевает, что функция является особенной или «принадлежит» какому-то другому объекту.

Возможно, кто-то возразит, что функция становится методом не в момент объявления, а в процессе выполнения этого конкретного вызова, в зависимости от того как она была вызвана в её точке вызова (в контексте ссылки на объект или нет -- подробнее смотрите в Главе 2). Даже такая интерпретация выглядит притянутой за уши.

Возможно, самым безопасным выводом будет такой: «функция» и «метод» взаимозаменяемы в JavaScript.

Примечание: ES6 добавляет указатель `super`, который обычно используется вместе с `class` (см. Приложение A). То, как работает `super` (статическая привязка вместо поздней привязки в виде `this`) прибавляет весомости идеи, что функция, которую привязывает `super` больше похожа на «метод», чем на «функцию». Но опять же, это лишь тонкие нюансы семантики (и механики).

Даже если вы объявляете функциональное выражение как часть литерала объекта, эта функция не становится волшебным образом более привязанной к объекту -- это по-прежнему лишь несколько указателей на ту же функцию.

```
var myObject = {
  foo: function foo() {
    console.log( "foo" );
  }
};

var someFoo = myObject.foo;
someFoo;          // function foo(){..}
myObject.foo;     // function foo(){..}
```

Примечание: В Главе 6 мы рассмотрим сокращенный синтаксис ES6 для `foo: function foo(){..}` в нашем литерале объекта.

› Массивы

Массивы тоже используют для доступа форму [], но как упоминалось ранее, имеют более структурированную организацию того, как хранятся значения (хотя всё еще без ограничений *типов* хранимых значений). Массивы предполагают *числовую индексацию*. Это означает, что значения хранятся в местах, обычно называемых *индексами*, с неотрицательными целыми числами, вроде 0 и 42.

```
var myArray = [ "foo", 42, "bar" ];
myArray.length; // 3
myArray[0];      // "foo"
myArray[2];      // "bar"
```

Массивы -- это объекты, поэтому даже если каждый индекс является положительным целым числом, вы можете *также* добавить свойства массива:

```
var myArray = [ "foo", 42, "bar" ];
myArray.baz = "baz";
myArray.length; // 3
myArray.baz;    // "baz"
```

Обратите внимание, что добавление именованных свойств (не зависимо от синтаксиса . или []) не влияет на выводимый результат свойства length.

Вы можете использовать массив как простой объект вида ключ/значение и не добавлять числовые индексы, но это плохая идея, поскольку у массивов есть особенное поведение и оптимизации, заточенные для их использования; то же самое и с обычными объектами. Используйте объекты для хранения пар ключ/значение и массивы для хранения значений с числовыми индексами.

Осторожно: Если вы попытаетесь добавить свойство к массиву, но имя свойства *выглядит* как число, оно добавится в виде числового индекса (таким образом изменится содержимое массива):

```
var myArray = [ "foo", 42, "bar" ];
myArray["3"] = "baz";
myArray.length; // 4
myArray[3];     // "baz"
```

› Дублирование Объектов

Одна из наиболее часто запрашиваемых возможностей JavaScript для разработчиков, которые только знакомятся с языком, это копирование объекта. Казалось бы, должен быть простой встроенный метод copy(), верно? Оказывается, всё немного сложнее, потому что изначально не до конца понятно каким должен быть алгоритм для создания дубликата.

Для примера рассмотрим этот объект:

```

function anotherFunction() { /*...*/ }
var anotherObject = {
  c: true
};
var anotherArray = [];
var myObject = {
  a: 2,
  b: anotherObject, // ссылка, а не копия!
  c: anotherArray, // еще одна ссылка!
  d: anotherFunction
};
anotherArray.push( anotherObject, myObject );

```

Как же на самом деле должна выглядеть копия объекта `myObject`?

Во-первых, мы должны решить будет это *поверхностная* или *глубокая* копия?

Результатом *поверхностного копирования* в новом объекте будет свойство `a` в виде копии значения `2`, но свойства `b`, `c`, и `d` -- лишь в виде ссылки на те же места, что и в оригинальном массиве. *Глубокая копия* продублирует не только `myObject`, но и `anotherObject` и `anotherArray`. Но тогда у нас будет проблема с тем, что `anotherArray` содержит ссылки на `anotherObject` и `myObject`, так что их тоже нужно скопировать вместо того чтобы сохранять ссылку. Теперь у нас есть проблема бесконечного дублирования из-за зацикленности ссылок.

Должны ли мы выявлять циклические ссылки и просто прерывать циклический обход (оставляя глубокие элементы не до конца продублированными)? Может просто вывести ошибку? Или что-то среднее?

Более того, не до конца ясно что будет означать *дублирование* функции. Существует несколько хаков, вроде вытягивания исходного кода функции через `toString()` (которые варьируются в разных реализациях и даже не являются надежными для всех движков, в зависимости от типа проверяемой функции)

Так как же нам решить все эти каверзные вопросы? Различные фреймворки имеют свои собственные интерпретации и решения. Но какое из них (если такое имеется) должен принять JS в качестве *стандартного*? Долгое время не было четкого ответа.

Одно из решений заключается в том, что объекты безопасные для JSON (то есть те, которые можно преобразовать в строку JSON и распарсить с теми же значениями и структурой) могут быть легко продублированы с помощью:

```
var newObj = JSON.parse( JSON.stringify( someObj ) );
```

Конечно, для этого вам нужно убедиться, что ваш объект безопасен для JSON. В некоторых ситуациях это элементарно. В других этого недостаточно.

В то же время, поверхностное копирование достаточно понятно и имеет меньше проблем, поэтому в ES6 для этой задачи есть `Object.assign(..)`. `Object.assign(..)` принимает целевой объект в качестве первого параметра, а также один или более исходных объектов в качестве последующих параметров. Он проходит по всем *перечисляемым* (см. ниже), *собственным ключам* (*существующим*

непосредственно) в исходном объекте(ах) и копирует их (только через присваивание `=`) в целевой объект. Кроме того, удобно, что он возвращает целевой объект, как показано ниже:

```
var newObj = Object.assign( {}, myObject );
newObj.a;                      // 2
newObj.b === anotherObject;     // true
newObj.c === anotherArray;      // true
newObj.d === anotherFunction;   // true
```

Примечание: В следующем разделе мы опишем «дескрипторы свойств» (характеристики свойств) и покажем использование `Object.defineProperty(..)`. Как бы то ни было, дублирование, которое имеет место в `Object.assign(..)` это чистое присваивание в стиле `=`, так что любые особенные характеристики свойств (вроде `writable`) исходного объекта *не сохраняются* в целевом объекте.

Дескрипторы свойств

Вплоть до ES5 язык JavaScript не давал вашему коду напрямую проверить или описать различия между характеристиками свойств: например, узнать доступно ли свойство только для чтения или нет.

Но в ES5 все свойства описываются с помощью **дескриптора свойств**.

Рассмотрим такой код:

```
var myObject = {
  a: 2
};
Object.getOwnPropertyDescriptor( myObject, "a" );
// {
// value: 2,
// writable: true,
// enumerable: true,
// configurable: true
// }
```

Как видно, дескриптор свойства (называемый «дескриптором данных» поскольку он хранит только значение данных) для нашего обычного свойства `a` это больше чем просто `его value, равное 2.`

Поскольку мы знаем значения по умолчанию для характеристик дескриптора свойств при создании обычного свойства, мы можем использовать `Object.defineProperty(..)` для добавления нового или изменения существующего свойства (если оно является `configurable!`) с желаемыми характеристиками.

Например:

```
var myObject = {};
Object.defineProperty( myObject, "a", {
```

```

    value: 2,
    writable: true,
    configurable: true,
    enumerable: true
} );
myObject.a; // 2

```

С помощью `defineProperty(..)` мы вручную добавили простое, обычное свойство `a` к объекту `myObject` в явном виде. Как бы то ни было, в общем случае вам не придется использовать ручной способ, пока вы не захотите изменить обычное поведение характеристик дескриптора.

› Перезаписываемое

Возможность изменить значение свойства контролируется характеристикой `writable`.

Рассмотрим:

```

var myObject = {};
Object.defineProperty( myObject, "a", {
    value: 2,
    writable: false, // не перезаписываемо!
    configurable: true,
    enumerable: true
} );
myObject.a = 3;
myObject.a;           // 2

```

Как видите, наша попытка модифицировать `value` не удалась, молча. В строгом режиме `strict mode` мы получим ошибку:

```

"use strict";
var myObject = {};
Object.defineProperty( myObject, "a", {
    value: 2,
    writable: false, // не перезаписываемо!
    configurable: true,
    enumerable: true
} );
myObject.a = 3;      // TypeError

```

`TypeError` говорит о том, что мы не можем изменить неперезаписываемое свойство

Примечание: Мы скоро обсудим геттеры и сеттеры, но вкратце, вы можете заметить, что `writable:false` означает, что значение нельзя изменить. Это отчасти равносильно указанию NOOP-сеттера. На самом деле, чтобы действительно соответствовать `writable:false` ваш NOOP-сеттер при вызове должен выдавать `TypeError`.

› Конфигурируемое

Пока свойство является конфигурируемым, мы можем изменять описание дескриптора, используя всё тот же инструмент `defineProperty(..)`.

```
var myObject = {
  a: 2
};
myObject.a = 3;
myObject.a;      // 3
Object.defineProperty( myObject, "a", {
  value: 4,
  writable: true,
  configurable: false,           // не конфигурируемо!
  enumerable: true
} );
myObject.a;      // 4
myObject.a = 5;
myObject.a;      // 5
Object.defineProperty( myObject, "a", {
  value: 6,
  writable: true,
  configurable: true,
  enumerable: true
} );          // TypeError
```

Последний вызов `defineProperty(..)` приводит к ошибке `TypeError`, вне зависимости от strict mode, если вы пытаетесь изменить значение дескриптора неконфигурируемого свойства. Осторожно: как видите, изменение `configurable` на `false` необратимо и его нельзя отменить.

Примечание: существует особенное исключение, о котором стоит помнить: если для свойства уже задано `configurable:false`, то `writable` может быть изменено с `true` на `false` без ошибки, но не обратно в `true` если оно уже `false`.

А еще `configurable:false` препятствует возможности использовать оператор `delete` для удаления существующего свойства.

```
var myObject = {
  a: 2
};
myObject.a;      // 2
delete myObject.a;
myObject.a;      // undefined
Object.defineProperty( myObject, "a", {
  value: 2,
  writable: true,
  configurable: false,
  enumerable: true
} );
myObject.a;      // 2
delete myObject.a;
myObject.a;      // 2
```

Как видите, последний вызов `delete` не удался (молча), потому что мы сделали свойство `a` неконфигурируемым.

`delete` используется только для удаления свойств объекта (которое может быть удалено) напрямую из указанного объекта. Если свойство объекта -- это последняя оставшаяся ссылка на некоторый объект/функцию и вы удаляете его, то ссылка удалится и теперь не имеющий ссылок объект/функция могут быть убраны сборщиком мусора.

› Перечисляемое

Последнее свойство дескриптора, о котором мы расскажем (есть еще два других, с которыми мы будем иметь дело, когда обсудим геттеры/сеттеры), это `enumerable`.

Возможно, это очевидно из названия, но этот параметр указывает, появится ли свойство в определенных перечислениях свойств объекта, таких как цикл `for..in`. Установите `false`, чтобы свойство не появлялось в подобных перечислениях, даже если оно по-прежнему полностью доступно. Установите `true`, чтобы оно присутствовало.

Все нормальные свойства, заданные пользователем, по умолчанию являются `enumerable`, поскольку обычно это то, что вам нужно. Но если у вас есть особенное свойство, которое вы хотите спрятать от перечислений, установите для него `enumerable:false`.

Мы скоро продемонстрируем перечисляемость более подробно, так что поставьте в уме закладку на эту тему.

› Иммутабельность

Иногда требуется создать свойства или объекты, которые не могут быть изменены (случайно или преднамеренно). ES5 добавляет для работы с этим несколько различных способов с определенными тонкостями.

Важно отметить, что **всё** это -- попытки создать неглубокую иммутабельность. Они влияют только на объект и характеристики его непосредственных свойств. Если объект содержит указатель на другой объект (массив, объект, функцию и т.д.), то *содержимое* другого объекта не будет затронуто и останется изменяемым.

```
myImmutableObject.foo; // [1,2,3]
myImmutableObject.foo.push( 4 );
myImmutableObject.foo; // [1,2,3,4]
```

В этом фрагменте мы предполагаем, что `myImmutableObject` уже создан и защищен как иммутабельный. Но чтобы также защитить содержимое `myImmutableObject.foo` (которое само по себе является объектом-массивом), вам также нужно сделать иммутабельным `foo`, используя один или несколько следующих способов.

Примечание: Нет ничего страшного в создании глубоко укоренившихся неизменяемых объектов в программах на JS. Особые случаи определённо требуют этого, но в качестве общего шаблона проектирования, если вы обнаружите у себя желание запечатать или заморозить все объекты, вы можете сделать шаг назад и

пересмотреть структуру программы, чтобы сделать её более устойчивой к возможным изменениям в значениях объектов.

› Константа объекта

Комбинируя `writable:false` и `configurable:false` вы по сути можете создать константу (не может быть изменена, переопределена или удалена) в качестве свойства объекта, вроде:

```
var myObject = {};
Object.defineProperty( myObject, "FAVORITE_NUMBER", {
  value: 42,
  writable: false,
  configurable: false
} );
```

› Запрет расширения

Если вы хотите запретить добавление новых свойств объекта, но в то же время оставить существующие свойства нетронутыми, используйте `Object.preventExtensions(..)`

```
var myObject = {
  a: 2
};
Object.preventExtensions( myObject );
myObject.b = 3;
myObject.b; // undefined
```

В нестрогом режиме, создание `b` завершится неудачей без ошибок. В строгом режиме это приведет к ошибке `TypeError`.

› Запечатывание

Метод `Object.seal(..)` создает «запечатанный» объект -- то есть принимает существующий объект и по сути применяет к нему `Object.preventExtensions(..)`, но также помечает все существующие свойства как `configurable:false`.

Таким образом, вы не можете не только добавлять свойства, но и переконфигурировать или удалить существующие (хотя вы всё еще можете изменять их значения).

› Заморозка

Метод `Object.freeze(..)` создает замороженный объект, что означает, что он принимает существующий объект и по сути применяет к нему `Object.seal(..)`, но также помечает все свойства «доступа к данным» как `writable:false`, так, что их значения не могут быть изменены.

Этот подход дает наивысший уровень иммутабельности, который вы можете получить для самого объекта, поскольку он предотвращает любые изменения в объекте или его

непосредственных свойствах (хотя, как сказано выше, содержимое любых других привязанных объектов не затрагивается).

Вы можете «глубоко заморозить» объект, применив `Object.freeze(..)` к объекту и рекурсивно перебрать все объекты, на которые он ссылается (которые еще не были затронуты) применив к ним `Object.freeze(..)`. Однако, будьте осторожны, поскольку это может затронуть другие (общие) объекты, которые вы не планировали менять.

› [[Get]]

Есть одна маленькая, но важная деталь, связанная с тем, как происходит доступ к свойствам.

Рассмотрим:

```
var myObject = {
  a: 2
};
myObject.a; // 2
```

`myObject.a` -- это запрос свойства, но он не *просто* ищет в `myObject` свойство с именем `a`, как может показаться.

Согласно спецификации, код выше выполняет операцию `[[Get]]` (что-то вроде вызова функции `[[Get]]()`) с объектом `myObject`. Стандартная встроенная операция `[[Get]]` проверяет объект на наличие запрашиваемого свойства и если находит его, то возвращает соответствующее значение.

Однако, в алгоритме `[[Get]]` описано важное поведение для случая, когда она *не* находит запрошенное свойство. В главе 5 мы узнаем что происходит дальше (обход по цепочке `[[Prototype]]`, если что).

Но один из важных результатов операции `[[Get]]` заключается в том, что если она по какой-либо причине не может найти значение запрошенного свойства, то вернёт значение `undefined`.

```
var myObject = {
  a: 2
};
myObject.b; // undefined
```

Это поведение отличается от случая, когда вы обращаетесь к *переменным* по имени их идентификатора. Если вы запросите переменную, которая не может быть найдена с помощью поиска по лексической области видимости, то результатом будет не `undefined`, как у свойств объекта, а ошибка `ReferenceError`.

```
var myObject = {
  a: undefined
};
```

```
myObject.a; // undefined  
myObject.b; // undefined
```

С точки зрения **значения**, нет разницы между этими двумя вызовами -- они оба выдадут `undefined`. Однако, внутри операции `[[Get]]`, хоть это и не заметно на первый взгляд, потенциально выполняется немного больше «работы» для вывода `myObject.b`, чем для вывода `myObject.a`.

Проверяя лишь результаты вывода значения, вы не можете отличить когда существует свойство, явно содержащее значение `undefined`, а когда свойство *не* существует и `undefined` -- это значение, по умолчанию возвращаемое, если `[[Get]]` не может вернуть нечто определённое.

› **[[Put]]**

Поскольку существует встроенная операция `[[Get]]` для получения значения свойства, очевидно, должна существовать и стандартная операция `[[Put]]`.

Заманчиво думать, что назначение свойства объекту просто вызовет `[[Put]]` чтобы задать или создать это свойство для запрашиваемого объекта. Но ситуация сложнее, чем кажется.

Поведение `[[Put]]` при вызове зависит от нескольких факторов, включая (наиболее значимый): существует ли такое свойство у объекта или нет.

Если свойство существует, то алгоритм `[[Put]]` проверит примерно следующее:

1. Является ли свойство дескриптором доступа (смотрите раздел «Геттеры и Сеттеры» ниже)? **Если да, то вызовет сеттер, если он есть.**
2. Является ли свойство дескриптором данных с ключом `writable` равным `false`? **Если да, то тихо завершится в нестрогом режиме `[non-strict mode]`, или выдаст ошибку `TypeError` в строгом режиме `[strict mode]`.**
3. Иначе, установит значение существующего свойства как обычно.

Если свойство запрашиваемого объекта еще не задано, то операция `[[Put]]` еще более сложная и запутанная. Мы вернемся к этому сценарию в Главе 5, когда обсудим `[[Prototype]]`, чтобы внести больше ясности.

› **Геттеры и Сеттеры**

Стандартные операции объектов `[[Put]]` и `[[Get]]` полностью контролируют как значения задаются для существующих или новых свойств и, соответственно, запрашиваются из существующих свойств.

Примечание: При использовании будущих/расширенных возможностей языка можно переопределить стандартные операции `[[Get]]` или `[[Put]]` для всего объекта (а не только для свойства). Данная тема выходит за рамки обсуждения этой книги, но будет охвачена позже в серии «Вы не знаете JS».

ES5 представил способ переопределения части этих стандартных операций не на уровне объекта, а на уровне свойств, через использование геттеров и сеттеров. Геттеры -- это

свойства, которые на самом деле вызывают скрытую функцию для получения значения. Сеттеры -- это свойства, которые на самом деле вызывают скрытую функцию для задания значения.

Когда вы задаете свойству геттер или сеттер, оно определяется как «дескриптор доступа» (в противовес «дескриптору данных»). Для дескрипторов доступа, характеристики дескриптора `value` и `writable` игнорируются, а вместо этого JS рассматривает характеристики свойства `set` и `get` (а также `configurable` и `enumerable`).

Рассмотрим:

```
var myObject = {
    // определяем геттер для `a`
    get a() {
        return 2;
    }
};

Object.defineProperty(
    myObject, // цель
    "b", // имя свойства
    { // дескриптор
        // определяем геттер для `b`
        get: function(){ return this.a * 2 },
        // убедимся что `b` будет отображаться как свойство объекта
        enumerable: true
    }
);
myObject.a; // 2
myObject.b; // 4
```

Как в объектно-литеральном синтаксисе с использованием `get a() { .. }`, так и с помощью явного определения через `defineProperty(..)` мы создали свойство объекта, которое на самом деле не содержит значение, но доступ к которому приводит к вызову функции-геттера, чьё возвращаемое значение и будет результатом обращения к свойству.

```
var myObject = {
    // определяем геттер для `a`
    get a() {
        return 2;
    }
};
myObject.a = 3;
myObject.a; // 2
```

Поскольку мы определили геттер для `a`, то если мы попытаемся установить значение `a`, операция не выдаст ошибки, а молча отбросит присваивание. Даже если бы тут был валидный сеттер, в нашем геттере жестко прописано вернуть только 2, так что операция присваивания будет спорной.

Чтобы этот сценарий более разумным, свойства должны быть заданы с помощью сеттеров, которые переопределяют стандартную операцию `[[Put]]` (известную как присваивание) для каждого свойства, как вы того и ожидали. Скорее всего вы захотите всегда объявлять и геттер и сеттер (наличие только первого или второго часто приводит к непредсказуемому/удивительному поведению):

```
var myObject = {
    // определим геттер для `a`
    get a() {
        return this._a_;
    },
    // определим сеттер для `a`
    set a(val) {
        this._a_ = val * 2;
    }
};
myObject.a = 2;
myObject.a; // 4
```

Примечание: В этом примере мы на самом деле сохраняем указанное значение присваивания 2 (операция `[[Put]]`) в другой переменной `_a_`. Имя `_a_` здесь чисто для примера и не означает никакого особенного поведения -- это обычное свойство, как и любое другое.

Существование

Ранее мы показали, что запрос свойства вроде `myObject.a` может вывести значение `undefined` как в случае, когда там явно задано `undefined`, так и в случае, когда свойство `a` вообще не существует. Если в обоих случаях значение одинаково, как же нам их различить?

Мы можем спросить есть ли у объекта свойство, не запрашивая значение свойства:

```
var myObject = {
    a: 2
};
("a" in myObject);           // true
("b" in myObject);           // false
myObject.hasOwnProperty( "a" ); // true
myObject.hasOwnProperty( "b" ); // false
```

Оператор `in` проверит находится ли свойство в объекте или существует ли оно уровнем выше в цепочке `[[Prototype]]` объекта (смотрите Главу 5). `hasOwnProperty(..)` наоборот проверяет есть ли свойство только у объекта `myObject` или нет и не опрашивает цепочку `[[Prototype]]`. Мы еще вернёмся к важным различиям между этими двумя операциями в Главе 5, когда исследуем `[[Prototype]]` более подробно.

Метод `hasOwnProperty(..)` доступен для всех нормальных объектов через делегирование `Object.prototype` (см. Главу 5). Но можно создать объект, который не привязан

к `Object.prototype` (с помощью `Object.create(null)` -- см. Главу 5). В этом случае, вызвать метод `myObject.hasOwnProperty(..)` не получится.

При таком сценарии более надежным способом выполнить подобную проверку будет `Object.prototype.hasOwnProperty.call(myObject, "a")`, который заимствует базовый метод `hasOwnProperty` и использует явную привязку `this` (см. Главу 2), чтобы применить его к нашему `myObject`.

Примечание: Оператор `in` выглядит так, будто он проверяет существование значения внутри контейнера, но на самом деле он проверяет существование имени свойства. Это отличие важно учитывать применительно к массивам, поскольку велик соблазн сделать проверку вроде `4 in [2, 4, 6]`, но она не будет вести себя так, как вы ожидали.

› Перечисление

Ранее мы кратко объяснили идею «перечисляемости», когда рассматривали `enumerable` -- характеристику дескриптора свойства. Давайте вернемся и рассмотрим её более подробно.

```
var myObject = { };
Object.defineProperty(
    myObject,
    "a",
    // сделаем `a` перечисляемой, как обычно
    { enumerable: true, value: 2 }
);
Object.defineProperty(
    myObject,
    "b",
    // сделаем `b` НЕперечисляемой
    { enumerable: false, value: 3 }
);
myObject.b;                      // 3
("b" in myObject);              // true
myObject.hasOwnProperty("b"); // true
// .....
for (var k in myObject) {
    console.log(k, myObject[k]);
}
// "a" 2
```

Вы заметите, что `myObject.b` по факту существует и имеет доступное значение, но оно не отображается в цикле `for..in` (хотя, внезапно, оно обнаружилось проверкой на существование оператором `in`). Всё потому, что по сути «перечислимое» означает «будет учтено, если пройти перебором по свойствам объекта»).

Примечание: Использование циклов `for..in` с массивами может выдать неожиданный результат, поскольку перечисление массива будет включать не только все численные индексы, но также перечисляемые свойства. Хорошая идея использовать циклы `for..in` только с объектами, а традиционные циклы `for` для перебора по численным индексам значений, хранящихся в массивах.

Еще один способ определить перечисляемые и неперечисляемые свойства:

```
var myObject = { };
Object.defineProperty(
  myObject,
  "a",
  // сделаем `a` перечисляемым, как обычно
  { enumerable: true, value: 2 }
);
Object.defineProperty(
  myObject,
  "b",
  // сделаем `b` неперечисляемым
  { enumerable: false, value: 3 }
);
myObject.propertyIsEnumerable( "a" );    // true
myObject.propertyIsEnumerable( "b" );    // false
Object.keys( myObject );                // ["a"]
Object.getOwnPropertyNames( myObject ); // ["a", "b"]
```

`propertyIsEnumerable(..)` проверяет существует ли данное имя свойства *непосредственно* в объекте и установлено ли `enumerable:true`.

`Object.keys(..)` возвращает массив всех перечисляемых свойств, в то время как `Object.getOwnPropertyNames(..)` возвращает массив *всех* свойств -- перечисляемых или нет.

Отличия `in` от `hasOwnProperty(..)` в том, опрашивают ли они цепочку `[[Prototype]]` или нет. В то время как `Object.keys(..)` и `Object.getOwnPropertyNames(..)` проверяют только конкретный указанный объект.

Не существует (пока) встроенного способа получить список **всех** свойств, эквивалентного тому, как опрашивает оператор `in` (перебирая все свойства по всей цепочке `[[Prototype]]`), как описано в Главе 5). Приблизительно, такой инструмент можно сделать, если рекурсивно перебирать цепочку `[[Prototype]]` объекта и на каждом уровне выбирать список из `Object.keys(..)` -- только перечисляемых свойств.

Итерация

Цикл `for..in` проходит по списку перечисляемых свойств объекта (включая его цепочку `[[Prototype]]`). Но что если вместо этого вы хотите перебрать именно значения?

В массивах с числовыми индексами перебор значений обычно выполняется стандартным циклом `for`, вроде:

```
var myArray = [1, 2, 3];
for (var i = 0; i < myArray.length; i++) {
  console.log( myArray[i] );
}
// 1 2 3
```

Это не перебор значений, а перебор индексов, где вы используете индекс для получения значения, наподобие `myArray[i]`.

ES5 добавил несколько вспомогательных итераторов для массивов, включая `forEach(..)`, `every(..)`, и `some(..)`. Каждый из этих помощников принимает функцию обратного вызова для каждого элемента массива. Отличия только в том, как они реагируют на значение, возвращаемое этой функцией.

`forEach(..)` перебирает все значения массива и игнорирует любые значения, возвращаемые функцией обратного вызова. `every(..)` продолжает перебор до конца или пока функция не вернёт `false` (или «ложное» значение), в то время как `some(..)` продолжает до конца или пока функция не вернёт значение `true` (или «истинное» значение).

Эти специальные возвращаемые значения внутри `every(..)` и `some(..)` действуют наподобие инструкции `break` внутри обычного цикла, поскольку они прекращают перебор задолго до конца.

Если вы перебираете объект циклом `for..in`, вы также лишь косвенно запрашиваете значения, поскольку он на самом деле перебирает только перечисляемые свойства объекта, заставляя вас обращаться к свойствам вручную для получения значений.

Примечание: В противовес перебору индексов массива в числовой последовательности (циклы `for` или другие итераторы), порядок перебора свойств объекта *не гарантирован* и может различаться в разных движках JS. Не полагайтесь на любую наблюдаемую последовательность для всего, что требует постоянства окружения, поскольку любое наблюдаемое ненадежно.

Но что если вместо индексов массива (или свойств объекта) вы хотите перебрать значения напрямую? К счастью, ES6 добавляет синтаксис цикла `for..of` для перебора массивов (и объектов, если объект определяет свой собственный итератор).

```
var myArray = [ 1, 2, 3 ];
for (var v of myArray) {
  console.log( v );
}
// 1
// 2
// 3
```

Цикл `for..of` запрашивает объект-итератор (из стандартной встроенной функции, на языке спецификации известной как `@@iterator`) у перебираемой сущности, а затем перебирает возвращаемые значения, вызывая метод `next()` объекта-итератора для каждой итерации цикла.

Массивы имеют встроенный `@@iterator`, поэтому `for..of` легко работает с ними, как показано выше. Давайте переберем массив вручную, используя встроенный `@@iterator`, чтобы посмотреть как он работает:

```
var myArray = [ 1, 2, 3 ];
var it = myArray[Symbol.iterator]();
```

```
it.next(); // { value:1, done:false }
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // { done:true }
```

Примечание: В `@@iterator` мы получаем *внутреннее свойство* объекта, используя `Symbol` из ES6: `Symbol.iterator`. Мы уже упоминали семантику `Symbol` ранее в этой главе (см. «Вычисляемые имена свойств»), поэтому здесь применяются те же рассуждения. Как правило, вы захотите обращаться к таким особенным свойствам через имя `Symbol`, а не через специальное значение, которое оно может содержать. Не смотря на подтекст в названии, `@@iterator` является **не объектом-итератором, а функцией, возвращающей объект-итератор** -- маленькая, но важная деталь.

Как показывает фрагмент выше, значение, которое возвращает вызов `next()` итератора, -- это объект вида `{ value: ... , done: ... }`, где `value` -- это значение текущей итерации, а `done` -- это `boolean`, показывающее, остались ли элементы для перебора.

Обратите внимание, что значение з вернулось вместе с `done:false`, что на первый взгляд может показаться странным. Вам нужно вызвать `next()` четвертый раз (что автоматически делает `for..of` из предыдущего фрагмента) чтобы получить `done:true` и понять, что вы действительно закончили перебор. Причина такого костыля находится за рамками текущего обсуждения, но она исходит из семантики генерирующих функций стандарта ES6.

В то время как массивы автоматически перебираются циклами `for..of`, обычные объекты **не имеют встроенного `@@iterator`**. Причины такого намеренного упущения намного сложнее чем то, что мы рассмотрим. В общих чертах, правильным было решение не добавлять реализацию, которая будет проблемной для будущих типов объектов.

Для перебора любого объекта можно определить свой собственный стандартный `@@iterator`. Например:

```
var myObject = {
  a: 2,
  b: 3
};
Object.defineProperty( myObject, Symbol.iterator, {
  enumerable: false,
  writable: false,
  configurable: true,
  value: function() {
    var o = this;
    var idx = 0;
    var ks = Object.keys( o );
    return {
      next: function() {
        return {
          value: o[ks[idx++]],
          done: (idx > ks.length)
        };
      }
    };
  }
}
```

```

} );
// перебираем `myObject` вручную
var it = myObject[Symbol.iterator]();
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // { value:undefined, done:true }
// перебираем `myObject` с помощью `for..of`
for (var v of myObject) {
    console.log( v );
}
// 2
// 3

```

Примечание: Мы использовали `Object.defineProperty(..)` чтобы задать свой `@@iterator` (в основном для того, чтобы сделать его неперечисляемым), но, используя `Symbol` как *рассчитанное имя свойства* (описанное ранее в этой главе), мы могли бы объявить его напрямую, вроде `var myObject = { a:2, b:3, [Symbol.iterator]: function(){ /* .. */ } }.`

Каждый раз, когда цикл `for..of` вызовет `next()` из итератора объекта `myObject`, внутренний указатель переместится и вернёт следующее значение из списка свойств объекта (см. примечание о порядке перебора свойств/значений объекта).

Мы продемонстрировали простой перебор «значение за значением», но вы, конечно, можете задать перебор произвольной сложности для своих структур данных так, как вам будет удобней. Самодельные итераторы вкупе с циклом `for..of` из ES6 -- мощный инструмент для работы с объектами, определяемыми пользователем.

Например, для списка объектов `Pixel` (со значениями координат `x` и `y`) можно упорядочить перебор в зависимости от линейного расстояния до начала координат `(0,0)` или отфильтровать точки, которые расположены «слишком далеко». Пока ваш итератор возвращает предполагаемое `{ value: .. }`, возвращает значения при вызове `next()` и `{ done: true }`, когда перебор завершен, цикл `for..of` стандарта ES6 сможет выполнить перебор.

Фактически, вы можете сгенерировать «бесконечные» итераторы, которые никогда не «завершатся» и всегда будут возвращать новое значение (вроде случайного числа, инкрементированного значения, уникального идентификатора и т.д.), хотя, скорее всего, вы не захотите использовать такие итераторы в неограниченном цикле `for..of`, поскольку он никогда не закончится и повесит вашу программу.

```

var randoms = {
    [Symbol.iterator]: function() {
        return {
            next: function() {
                return { value: Math.random() };
            }
        };
    }
};
var randoms_pool = [];
for (var n of randoms) {
    randoms_pool.push( n );
}

```

```
// не продолжаем бесконечно!
if (randoms_pool.length === 100) break;
}
```

Этот итератор будет генерировать случайные числа «вечно», поэтому мы позабочились о том, чтобы получить только 100 значений и наша программа не зависла.

’ Обзор (TL;DR)

Объекты в JS имеют литеральную форму (вроде `var a = { .. }`) и конструкторную форму (вроде `var a = new Array(..)`). Литеральная форма почти всегда предпочтительнее, но конструкторная форма в некоторых случаях предлагает больше опций при создании.

Многие ошибочно заявляют, что «в JS всё является объектом», но это некорректно. Объекты -- это один из 6 (или 7, в зависимости от ваших взглядов) примитивных типов. Существуют подтипы объектов, в том числе `function`, а также подтипы со специальным поведением, наподобие `[object Array]`, представляющего внутреннее обозначение такого подтипа объекта, как массив.

Объекты -- это коллекции ключ-значение. Значения могут быть получены через свойства, посредством синтаксиса `.propName` или `["propName"]`. Вне зависимости от синтаксиса, движок вызывает встроенную стандартную операцию `[[Get]]` (и `[[Put]]` для установки значений), которая не только ищет свойство непосредственно в объекте, но и перемещается по цепочке `[[Prototype]]` (см. Главу 5), если свойство не найдено.

У свойств есть определенные характеристики, которыми можно управлять через дескрипторы свойств, такие как `writable` и `configurable`. В дополнение, мутабельностью объектов (и их свойств) можно управлять на разных уровнях иммутабельности, используя `Object.preventExtensions(..)`, `Object.seal(..)`, и `Object.freeze(..)`.

Свойства не обязательно содержат значения -- они могут быть также «свойствами доступа» с геттерами/сеттерами. Они могут быть *перечисляемыми* или нет, что влияет на их появление в итерациях цикла, например `for..in`.

Вы также можете перебирать **значения** структур данных (массивов, объектов и т.п.) используя синтаксис ES6 `for..of`, который ищет встроенный или самодельный объект `@@iterator`, содержащий метод `next()` для перебора значений по одному.

Глава 5: Прототипы

В главах 3 и 4 мы неоднократно упоминали цепочку `[[Prototype]]`, но не уточняли что это такое. Пришло время разобраться с тем, как работают прототипы.

Примечание: Любые попытки эмуляции копирования классов, упомянутые в главе 4 как "примеси", полностью обходят механизм цепочки `[[Prototype]]`, рассматриваемый в этой главе.

› `[[Prototype]]`

Объекты в JavaScript имеют внутреннее свойство, обозначенное в спецификации как `[[Prototype]]`, которое является всего лишь ссылкой на другой объект. Почти у всех объектов при создании это свойство получает `не-null` значение.

Примечание: Чуть ниже мы увидим, что объект может иметь пустую ссылку `[[Prototype]]`, хотя такой вариант встречается реже.

Рассмотрим пример:

```
var myObject = {
  a: 2
};

myObject.a; // 2
```

Для чего используется ссылка `[[Prototype]]`? В главе 3 мы изучили операцию `[[Get]]`, которая вызывается когда вы ссылаетесь на свойство объекта, например `myObject.a`. Стандартная операция `[[Get]]` сначала проверяет, есть ли у объекта собственное свойство `a`, если да, то оно используется.

Примечание: Прокси-объекты ES6 выходят за рамки этой книги (мы увидим их в одной из следующих книг серии!), но имейте в виду, что обсуждаемое нами стандартное поведение `[[Get]]` и `[[Put]]` неприменимо если используются `Proxy`.

Но нас интересует то, что происходит, когда `a` **отсутствует** в `myObject`, т.к. именно здесь вступает в действие ссылка `[[Prototype]]` объекта.

Если стандартная операция `[[Get]]` не может найти запрашиваемое свойство в самом объекте, то она следует по **ссылке** `[[Prototype]]` этого объекта.

```
var anotherObject = {
  a: 2
};

// создаем объект, привязанный к `anotherObject`
var myObject = Object.create( anotherObject );
```

```
myObject.a; // 2
```

Примечание: Чуть позже мы объясним что делает `Object.create(..)` и как он работает. Пока же считайте, что создается объект со ссылкой `[[Prototype]]` на указанный объект.

Итак, у нас есть `myObject`, который теперь связан с `anotherObject` через ссылку `[[Prototype]]`. Очевидно, что `myObject.a` на самом деле не существует, однако обращение к свойству выполнилось успешно (свойство нашлось в `anotherObject`) и действительно вернуло значение 2.

Если бы `a` не нашлось и в объекте `anotherObject`, то теперь уже его цепочка `[[Prototype]]` использовалась бы для дальнейшего поиска.

Этот процесс продолжается до тех пор, пока либо не будет найдено свойство с совпадающим именем, либо не закончится цепочка `[[Prototype]]`. Если по достижении конца цепочки искомое свойство *так и не будет* найдено, операция `[[Get]]` вернет `undefined`.

По аналогии с этим процессом поиска по цепочке `[[Prototype]]`, если вы используете цикл `for..in` для итерации по объекту, будут перечислены все свойства, достижимые по его цепочке (при условии, что они перечислимые — см. `enumerable` в главе 3). Если вы используете оператор `in` для проверки существования свойства в объекте, то `in` проверит всю цепочку объекта (независимо от *перечисляемости*).

```
var anotherObject = {
  a: 2
};

// создаем объект, привязанный к `anotherObject`
var myObject = Object.create(anotherObject);

for (var k in myObject) {
  console.log("найдено: " + k);
}
// найдено: a

("a" in myObject); // true
```

Итак, ссылки в цепочке `[[Prototype]]` используются одна за другой, когда вы тем или иным способом пытаетесь найти свойство. Поиск заканчивается при нахождении свойства или достижении конца цепочки.

› `Object.prototype`

Но где именно "заканчивается" цепочка `[[Prototype]]`?

В конце каждой *типовичної* цепочки `[[Prototype]]` находится встроенный объект `Object.prototype`. Этот объект содержит различные утилиты, используемые в JS повсеместно, поскольку все обычные (встроенные, не связанные с конкретной средой

исполнения) объекты в JavaScript "происходят от" объекта `Object.prototype` (иными словами, имеют его на вершине своей цепочки `[[Prototype]]`).

Некоторые утилиты этого объекта могут быть вам знакомы: `.toString()` и `.valueOf()`. В главе 3 мы видели еще одну: `.hasOwnProperty(..)`. Еще одна функция `Object.prototype`, о которой вы могли не знать, но узнаете далее в этой главе — это `.isPrototypeOf(..)`.

Установка и затенение свойств

Ранее в главе 3 мы отмечали, что установка свойств объекта происходит чуть сложнее, чем просто добавление к объекту нового свойства или изменение значения существующего свойства.

```
myObject.foo = "bar";
```

Если непосредственно у `myObject` есть обычное свойство доступа к данным с именем `foo`, то присваивание сводится к изменению значения существующего свойства.

Если непосредственно у `myObject` нет `foo`, то выполняется обход цепочки `[[Prototype]]` по аналогии с операцией `[[Get]]`. Если `foo` не будет найдено в цепочке, то свойство `foo` добавляется непосредственно к `myObject` и получает указанное значение, как мы того и ожидаем.

Однако если `foo` находится где-то выше по цепочке, то присваивание `myObject.foo = "bar"` может повлечь за собой более сложное (и даже неожиданное) поведение. Рассмотрим этот вопрос подробнее.

Если свойство с именем `foo` присутствует как у самого `myObject`, так и где-либо выше в цепочке `[[Prototype]]`, начинающейся с `myObject`, то такая ситуация называется **затенением**. Свойство `foo` самого `myObject` **затеняет** любые свойства `foo`, расположенные выше по цепочке, потому что поиск `myObject.foo` всегда находит свойство `foo`, ближайшее к началу цепочки.

Как уже отмечалось, затенение `foo` в `myObject` происходит не так просто, как может показаться. Мы рассмотрим три сценария присваивания `myObject.foo = "bar"`, когда `foo` **не** содержится непосредственно в `myObject`, а **находится выше по цепочке `[[Prototype]]`** объекта `myObject`:

1. Если обычное свойство доступа к данным (см. главу 3) с именем `foo` находится где-либо выше по цепочке `[[Prototype]]`, и не отмечено как только для чтения (`writable:false`), то новое свойство `foo` добавляется непосредственно в объект `myObject`, и происходит **затенение свойства**.
2. Если `foo` находится выше по цепочке `[[Prototype]]`, но отмечено как только для чтения (`writable:false`), то установка значения этого существующего свойства, равно как и создание затененного свойства у `myObject`, запрещены. Если код выполняется в `strict mode`, то будет выброшена ошибка, если нет, то попытка установить значение свойства будет проигнорирована. В любом случае, **затенения не происходит**.
3. Если `foo` находится выше по цепочке `[[Prototype]]` и является сеттером (см. главу 3), то всегда будет вызываться сеттер. Свойство `foo` не будет добавлено в `myObject`, сеттер `foo` не

будет переопределен.

Большинство разработчиков предполагает, что присваивание свойства ([[Put]]) всегда приводит к затенению, если свойство уже существует выше по цепочке [[Prototype]], но как видите это является правдой лишь в одной (№1) из трех рассмотренных ситуаций.

Если вы хотите затенить foo в случаях №2 и №3, то вместо присваивания = нужно использовать object.defineProperty(..) (см. главу 3) чтобы добавить foo в myObject.

Примечание: Ситуация №2 может показаться наиболее удивительной из трех. Наличие свойства только для чтениямешает нам неявно создать (затенить) свойство с таким же именем на более низком уровне цепочки [[Prototype]]. Причина такого ограничения по большей части кроется в желании поддержать иллюзию наследования свойств класса. Если представить, что foo с верхнего уровня цепочки наследуется (копируется) в myObject, то имеет смысл запретить изменение этого свойства foo в myObject. Но если перейти от иллюзий к фактам и согласиться с тем, что никакого наследования с копированием на самом деле не происходит (см. главы 4 и 5), то кажется немного странным, что myObject не может иметь свойство foo лишь потому, что у какого-то другого объекта есть неизменяемое свойство foo. Еще более странно то, что это ограничение действует только на присваивание =, но не распространяется на object.defineProperty(..).

Затенение при использовании методов приведет к уродливому явному псевдополиморфизму (см. главу 4) если вам потребуется делегирование между ними. Обычно затенение приносит больше проблем и сложностей, чем пользы, поэтому старайтесь избегать его если это возможно. В главе 6 вы увидите альтернативный шаблон проектирования, который наряду с другими вещами предполагает отказ от затенения в пользу более разумных альтернатив.

Затенение может даже произойти неявно, поэтому если вы хотите его избежать, будьте бдительны. Например:

```
var anotherObject = {
  a: 2
};

var myObject = Object.create( anotherObject );

anotherObject.a; // 2
myObject.a; // 2

anotherObject.hasOwnProperty( "a" ); // true
myObject.hasOwnProperty( "a" ); // false

myObject.a++; // ой, неявное затенение!

anotherObject.a; // 2
myObject.a; // 3

myObject.hasOwnProperty( "a" ); // true
```

Хотя может показаться, что выражение `myObject.a++` должно (через делегирование) найти и просто инкрементировать свойство `anotherObject.a`, вместо этого операция `++` соответствует выражению `myObject.a = myObject.a + 1`. В результате `[[Get]]` ищет свойство `a` через `[[Prototype]]` и получает текущее значение `2` из `anotherObject.a`, далее это значение увеличивается на `1`, после чего `[[Put]]` присваивает значение `3` новому затененному свойству `a` в `myObject`. Ой!

Будьте очень осторожны при работе с делегированными свойствами, пытаясь изменить их значение. Если вам нужно инкрементировать `anotherObject.a`, то вот единственно верный способ сделать это: `anotherObject.a++`.

› "Класс"

Вы уже могли задаться вопросом: "Зачем одному объекту нужна ссылка ссылка на другой объект?" Какой от этого толк? Это очень хороший вопрос, но сначала нам нужно выяснить, чем `[[Prototype]]` не является, прежде чем мы сможем понять и оценить то, чем он является, и какая от него польза.

В главе 4 мы выяснили, что в отличие от класс-ориентированных языков в JavaScript нет абстрактных шаблонов/схем объектов, называемых "классами". В JavaScript просто есть объекты.

На самом деле, JavaScript — практически уникальный язык, ведь пожалуй только он имеет право называться "объектно-ориентированным", т.к. относится к весьма немногочисленной группе языков, где объекты можно создавать напрямую, без классов.

В JavaScript классы не могут описывать поведение объекта (учитывая тот факт, что их вообще не существует!). Объект сам определяет собственное поведение. **Есть только объект.**

› Функции "классов"

В JavaScript есть специфическое поведение, которым долгие годы цинично злоупотребляли для создания подделок, внешне похожих на "классы". Рассмотрим этот подход более подробно.

Специфическое поведение "как бы классов" основано на одной странной особенности: у всех функций по умолчанию есть публичное, неперечислимое (см. главу 3) свойство, называемое `prototype`, которое указывает на некий объект.

```
function Foo() {  
    // ...  
}  
  
Foo.prototype; // { }
```

Этот объект часто называют "прототипом `Foo`", поскольку обращение к нему происходит через свойство с неудачно выбранным названием `Foo.prototype`. Как мы вскоре увидим, такая терминология обречена приводить людей в замешательство. Вместо этого, я буду называть его "объектом, ранее известным как прототип `Foo`". Ладно, шучу. Как насчет этого: "объект, условно называемый 'Foo точка prototype'"?

Как бы мы не называли его, что же это за объект?

Проще всего объяснить так: у каждого объекта, создаваемого с помощью вызова `new Foo()` (см. главу 2), ссылка `[[Prototype]]` будет указывать на этот объект "Foo точка prototype".

Проиллюстрируем на примере:

```
function Foo() {
  // ...
}

var a = new Foo();

Object.getPrototypeOf( a ) === Foo.prototype; // true
```

Когда `a` создается путем вызова `new Foo()`, одним из результатов (все четыре шага см. в главе 2) будет создание в внутренней ссылки `[[Prototype]]` на объект, на который указывает `Foo.prototype`.

Остановитесь на секунду и задумайтесь о смысле этого утверждения.

В класс-ориентированных языках множественные **копии** (или "экземпляры") создаются как детали, штампаемые на пресс-форме. Как мы видели в главе 4, так происходит потому что процесс создания экземпляра (или наследования от) класса означает "скопировать поведение из этого класса в физический объект", и это выполняется для каждого нового экземпляра.

Но в JavaScript такого копирования не происходит. Вы не создаете множественные экземпляры класса. Вы можете создать множество объектов, *связанных** ссылкой `[[Prototype]]` с общим объектом. Но по умолчанию никакого копирования не происходит, поэтому эти объекты не становятся полностью автономными и не соединенными друг с другом, напротив, они весьма **связаны**.

Вызов `new Foo()` создает новый объект (мы назвали его `a`), и **этот** новый объект `a` связан внутренней ссылкой `[[Prototype]]` с объектом `Foo.prototype`.

В результате получилось два объекта, связанных друг с другом. Вот и все. Мы не создали экземпляр класса. И мы уж точно не копировали никакого поведения из "класса" в реальный объект. Мы просто связали два объекта друг с другом.

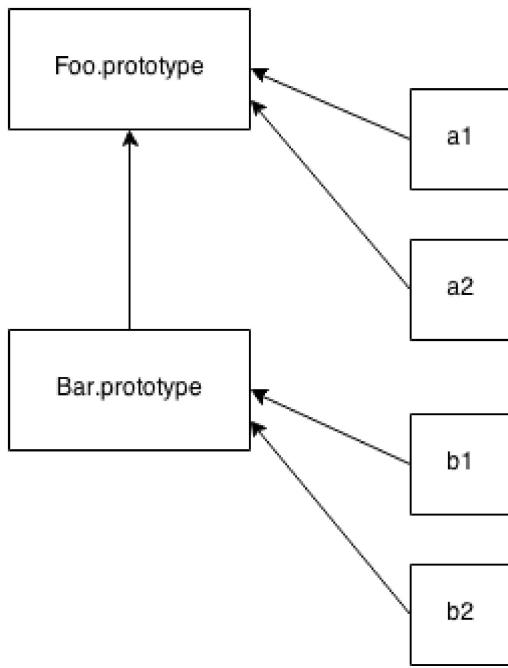
На самом деле секрет, о котором не догадывается большинство JS разработчиков, скроется в том, что вызов функции `new Foo()` практически никак *напрямую* не связан с процессом создания ссылки. Это всегда было неким побочным эффектом. `new Foo()` — это косвенный, окольный путь к желаемому результату: **новому объекту, связанному с другим объектом**.

Можем ли мы добиться желаемого более *прямым* путем? Да! Герой дня — `Object.create(..)`. Но мы вернемся к нему чуть позже.

› Что значит имя?

В JavaScript мы не делаем *копии* из одного объекта ("класса") в другой ("экземпляр"). Мы создаем *ссылки* между объектами. В механизме `[[Prototype]]` визуально стрелки идут справа

налево и снизу вверх.



Этот механизм часто называют "прототипным наследованием" (мы подробнее рассмотрим код чуть ниже), которое обычно считается вариантом "классического наследования" для динамических языков. Это попытка воспользоваться общепринятым пониманием термина "наследование" в класс-ориентированных языках и подогнать* знакомую семантику под динамический язык.

Термин "наследование" имеет очень четкий смысл (см. главу 4). Добавление перед ним слова "прототипное" чтобы обозначить *на самом деле почти противоположное поведение* привело к неразберихе в течение двух десятков лет.

Я люблю говорить, что использовать слово "прототипное" перед "наследованием" для придания существенно иного смысла — это как держать в одной руке апельсин, а в другой яблоко и настаивать на том, что яблоко — это "красный апельсин". Не важно, какое прилагательное я добавлю перед ним, это не изменит тот факт, что один фрукт — яблоко, а другой — апельсин.

Лучше называть вещи своими именами. Это позволяет лучше понять как их сходство, так и **многочисленные отличия**.

Из-за всей этой неразберихи с терминами я считаю, что само название "прототипное наследование" (а также некорректное использование связанных с ним терминов, таких как "класс", "конструктор", "экземпляр", "полиморфизм" и т.д.) принесло **больше вреда чем пользы** в понимании того, как *на самом деле* работает JavaScript.

"Наследование" подразумевает операцию *копирования*, а JavaScript не копирует свойства объекта (по умолчанию). Вместо этого JS создает ссылку между двумя объектами, в результате один объект по сути *делегирует* доступ к свойствам/функциям другому объекту. "*Делегирование*" (см. главу 6) — более точный термин для описания механизма связывания объектов в JavaScript.

Другой термин, который иногда встречается в JavaScript — это "дифференциальное наследование". Суть в том, что мы описываем поведение объекта с точки зрения того,

что отличается в нем от более общего описания. Например, вы можете сказать, что автомобиль является видом транспортного средства, у которого ровно 4 колеса, вместо того чтобы заново описывать все свойства транспортного средства (двигатель, и т.д.).

Если представить, что любой объект в JS является суммой всего поведения, которое доступно через делегирование, и мысленно объединить все это поведение в одну реальную сущность*, то можно предположить, что "дифференциальное наследование" (вроде как) подходящий термин.

Но как и "прототипное наследование", "дифференциальное наследование" претендует на то, что мысленная модель важнее того, что физически происходит в языке. Здесь упускается из виду факт, что объект в на самом деле не создается дифференциально, а создается с конкретно заданными характеристиками, а также с "дырами", где ничего не задано. Эти дыры (пробелы или отсутствующие определения) могут заменяться механизмом делегирования, который на лету "заполняет их" делегированным поведением.

Объект по умолчанию не сворачивается в единый дифференциальный объект **посредством копирования**, как это подразумевается в мысленной модели "дифференциального наследования". Таким образом, "дифференциальное наследование" не слишком подходит для описания реального механизма работы [[Prototype]] в JavaScript.

Вы можете* придерживаться терминологии и мысленной модели "дифференциального наследования", но нельзя отрицать тот факт, что это лишь упражнение ума в вашей голове, а не реальное поведение движка.

› "Конструкторы"

Вернемся к рассмотренному ранее коду:

```
function Foo() {
  // ...
}

var a = new Foo();
```

Что именно заставляет нас подумать, что Foo является "классом"?

С одной стороны, мы видим использование ключевого слова new, совсем как в класс-ориентированных языках, когда создаются экземпляры классов. С другой стороны, кажется, что мы на самом деле выполняем метод **конструктора** класса, потому что метод Foo() на самом деле вызывается, так же как конструктор реального класса вызывается при создании экземпляра.

У объекта Foo.prototype есть еще один фокус, который усиливает недоразумение, связанное с семантикой "конструкторов". Посмотрите на этот код:

```
function Foo() {
  // ...
}
```

```
Foo.prototype.constructor === Foo; // true
```

```
var a = new Foo();
a.constructor === Foo; // true
```

По умолчанию объект `Foo.prototype` (во время объявления в первой строке примера!) получает публичное неперечислимое (см. главу 3) свойство `.constructor`, и это свойство является обратной ссылкой на функцию (в данном случае `Foo`), с которой связан этот объект. Более того, мы видим, что объект `a`, созданный путем вызова "конструктора" `new Foo()`, похоже тоже имеет свойство с именем `.constructor`, также указывающее на "функцию, создавшую его".

Примечание: На самом деле это неправда. У `a` нет свойства `.constructor`, и хотя `a.constructor` действительно разрешается в функцию `Foo`, "конструктор" **на самом деле не значит** "был сконструирован этой функцией". Мы разберемся с этим курьезом чуть позже.

Ах, да, к тому же... в мире JavaScript принято соглашение об именовании "классов" с заглавной буквы, поэтому тот факт что ЭТО `Foo`, а не `foo` является четким указанием, что мы хотим определить "класс". Это ведь абсолютно очевидно, не так ли!?

Примечание: Это соглашение имеет такое влияние, что многие JS линтеры возмущаются когда вы вызываете `new` с методом, имя которого состоит из строчных букв, или не вызываете `new` с функцией, начинающейся с заглавной буквы. Удивительно, что мы с таким трудом пытаемся добиться (фальшивой) "класс-ориентированности" в JavaScript, что даже создаем правила для линтеров, чтобы гарантировать использование заглавных букв, хотя заглавные буквы **вообщеничего** не значат для движка JS.

› Конструктор или вызов?

В примере выше есть соблазн предположить, что `Foo` — это "конструктор", потому что мы вызываем её с `new` и видим, что она "конструирует" объект.

В действительности, `Foo` такой же "конструктор", как и любая другая функция в вашей программе. Функции сами по себе **не являются** конструкторами. Однако когда вы добавляете ключевое слово `new` перед обычным вызовом функции, это превращает вызов функции в "вызов конструктора". На самом деле `new` как бы перехватывает любую обычную функцию и вызывает её так, что в результате создается объект, а **также выполняется код самой функции**.

Например:

```
function NothingSpecial() {
    console.log( "Don't mind me!" );
}

var a = new NothingSpecial();
// "Don't mind me!"

a; // {}
```

`NothingSpecial` — обычная функция, но когда она вызывается с `new`, то практически в качестве побочного эффекта *создает* объект, который мы присваиваем `a`. Этот *вызов был вызовом конструктора*, но сама по себе функция `NothingSpecial` не является *конструктором*.

Иначе говоря, в JavaScript "конструктор" — это **любая функция, вызванная с ключевым словом `new` перед ней**.

Функции не являются конструкторами, но вызовы функций являются "вызовами конструктора" тогда и только тогда, когда используется `new`.

Механика

Являются ли эти особенности единственными причинами многострадальных дискуссий о "классах" в JavaScript?

Не совсем. JS разработчики постарались симулировать поведение классов настолько, насколько это возможно:

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

Foo.prototype.myName = function() {
    return this.name;
};

var a = new Foo( "a" );
var b = new Foo( "b" );

a.myName(); // "a"
b.myName(); // "b"
```

Этот пример показывает два дополнительных трюка для "класс-ориентированности":

1. `this.name = name`: свойство `.name` добавляется в каждый объект (`a` и `b`, соответственно; см. главу 2 о привязке `this`), аналогично тому как экземпляры классов инкапсулируют значения данных.
2. `Foo.prototype.myName = ...`: возможно более интересный прием, добавляет свойство (функцию) в объект `Foo.prototype`. Теперь работает `a.myName()`, но каким образом?

В примере выше велик соблазн думать, что при создании `a` и `b` свойства/функции объекта `Foo.prototype` копируются в каждый из объектов `a` и `b`. **Однако этого не происходит.**

В начале этой главы мы изучали ссылку `[[Prototype]]` — часть стандартного алгоритма `[[Get]]`, которая предоставляет запасной вариант поиска, если ссылка на свойство отсутствует в самом объекте.

В силу того, как создаются `a` и `b`, оба объекта получают внутреннюю ссылку `[[Prototype]]` на `Foo.prototype`. Когда `myName` не находится в `a` или `b` соответственно, она обнаруживается (через делегирование, см. главу 6) в `Foo.prototype`.

› И снова о "конструкторе"

Вспомните наше обсуждение свойства `.constructor`. Кажется, что `a.constructor === Foo` означает, что в `a` есть реальное свойство `.constructor`, указывающее на `Foo`, верно? Не верно.

Это всего лишь путаница. На самом деле ссылка `.constructor` также *делегируется* вверх по цепочке в `Foo.prototype`, у которого, **так уж случилось**, по умолчанию есть свойство `.constructor`, указывающее на `Foo`.

Кажется ужасно удобным, что у объекта `a`, "созданного" `Foo`, будет доступ к свойству `.constructor`, которое указывает на `Foo`. Но это ложное чувство безопасности. Лишь по счастливой случайности `a.constructor` указывает на `Foo` через делегирование `[[Prototype]]` по умолчанию. На самом деле есть несколько способов наломать дров, предполагая что `.constructor` означает "использовался для создания".

Начнем с того, что свойство `.constructor` в `Foo.prototype` по умолчанию есть лишь у объекта, создаваемого в момент объявления функции `Foo`. Если создать новый объект и заменить у функции ссылку на стандартный объект `.prototype`, то новый объект по умолчанию не получит свойства `.constructor`.

Рассмотрим:

```
function Foo() { /* .. */ }

Foo.prototype = { /* .. */ }; // создаем новый объект-прототип

var a1 = new Foo();
a1.constructor === Foo; // false!
a1.constructor === Object; // true!
```

`object(..)` не был "конструктором" `a1`, не так ли? Выглядит так, будто "конструктором" объекта должна быть `Foo()`. Многие разработчики думают, что `Foo()` создает объект, но эта идея трещит по швам, когда вы думаете что "constructor" значит "был создан при помощи". Ведь в таком случае `a1.constructor` должен указывать на `Foo`, но это не так!

Что же происходит? У `a1` нет свойства `.constructor`, поэтому он делегирует вверх по цепочке `[[Prototype]]` к `Foo.prototype`. Но и у этого объекта нет `.constructor` (в отличие от стандартного объекта `Foo.prototype!`), поэтому делегирование идет дальше, на этот раз до `Object.prototype` — вершины цепочки делегирования. У этого объекта действительно есть `.constructor`, который указывает на встроенную функцию `Object(..)`.

Разрушаем заблуждение.

Конечно, можно вернуть объекту `Foo.prototype` свойство `.constructor`, но это придется сделать вручную, особенно если вы хотите, чтобы свойство соответствовало стандартному поведению и было не перечислимым (см. главу 3).

Например:

```

function Foo() { /* .. */ }

Foo.prototype = { /* .. */ }; // создаем новый объект-прототип

// Необходимо правильно "пофиксить" отсутствующее свойство `constructor`
// у нового объекта, выступающего в роли `Foo.prototype`.
// См. главу 3 про `defineProperty(..)`.

Object.defineProperty( Foo.prototype, "constructor" , {
    enumerable: false,
    writable: true,
    configurable: true,
    value: Foo // `constructor` теперь указывает на `Foo`
} );

```

Как видите, для исправления `.constructor` необходимо много ручной работы. Больше того, все это мы делаем ради поддержания ошибочного представления о том, что "`constructor`" означает "используется для создания". Дорого же нам обходится эта иллюзия.

Факт в том, что `.constructor` объекта по умолчанию указывает на функцию, которая, в свою очередь, имеет обратную ссылку на объект — ссылку `.prototype`. Слова "конструктор" и "прототип" лишь наделяются по умолчанию ненадежным смыслом, который позднее может оказаться неверным. Лучше всего постоянно напоминать себе, что "конструктор не значит используется для создания".

`.constructor` — это не магическое неизменяемое свойство. Оно является, неперечисляемым (см. пример выше), но его значение доступно для записи (может быть изменено), и более того, вы можете добавить или перезаписать (намеренно или случайно) свойство с именем `constructor` в любом объекте любой цепочки `[[Prototype]]`, задав ему любое подходящее вам значение.

В силу того как алгоритм `[[Get]]` обходит цепочку `[[Prototype]]`, ссылка на свойство `.constructor`, найденная в любом узле цепочки, может получать значение, весьма отличающееся от ожидаемого.

Видите, насколько произвольным является его значение?

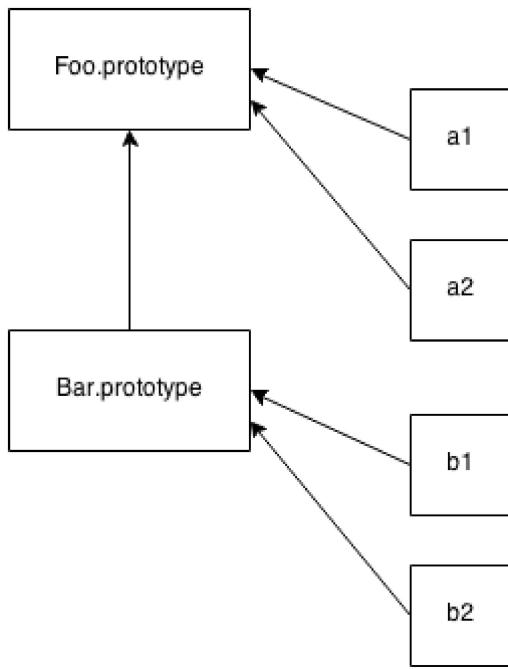
Что в итоге? Некая произвольная ссылка на свойство объекта, например `a1.constructor`, не может считаться надежной ссылкой на функцию по умолчанию. Более того, как мы вскоре увидим, в результате небольшого упущения `a1.constructor` может вообще указывать на весьма странное и бессмысленное значение.

`a1.constructor` — слишком ненадежная и небезопасная ссылка, чтобы полагаться на нее в коде. В общем случае таких ссылок по возможности следует избегать.

› "(Прототипное) наследование"

Мы увидели некоторые типичные хаки для добавления механики "классов" в программы на JavaScript. Но "классы" JavaScript были бы неполными без попыток смоделировать "наследование".

На самом деле мы уже видели механизм под названием "прототипное наследование", когда объект a "унаследовал от" Foo.prototype функцию myName(). Но обычно под "наследованием" подразумевается отношение между двумя "классами", а не между "классом" и "экземпляром".



Мы уже видели эту схему, на которой показано не только делегирование от объекта (или "экземпляра") a1 к объекту Foo.prototype, но и от Bar.prototype к Foo.prototype, что отчасти напоминает концепцию наследования классов родитель-потомок. Вот только направление стрелок здесь другое, поскольку изображены делегирующие ссылки, а не операции копирования.

Вот типичный пример кода в "прототипном стиле", где создаются такие ссылки:

```

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

Foo.prototype.myName = function() {
    return this.name;
};

function Bar(name,label) {
    Foo.call( this, name );
    this.label = label;
}

// здесь мы создаем `Bar.prototype`
// связанный с `Foo.prototype`
Bar.prototype = Object.create( Foo.prototype );

// Осторожно! Теперь `Bar.prototype.constructor` отсутствует,
// и это придется "пофиксить" вручную,
// если вы привыкли полагаться на подобные свойства!

Bar.prototype.myLabel = function() {
    return this.label;
}
  
```

```

};

var a = new Bar( "a", "obj a" );

a.myName(); // "a"
a.myLabel(); // "obj a"

```

Примечание: Чтобы понять, почему `this` указывает на `a`, см. главу 2.

Самая важная строка здесь `Bar.prototype = Object.create(Foo.prototype)`. `Object.create(..)` создает "новый" объект из ничего, и связывает внутреннюю ссылку `[[Prototype]]` этого объекта с указанным объектом (в данном случае `Foo.prototype`).

Другими словами, эта строка означает: "создать новый объект 'Bar точка prototype', связанный с 'Foo точка prototype'".

При объявлении `function Bar() { .. }` функция `Bar`, как и любая другая, получает ссылку `.prototype` на объект по умолчанию. Но этот объект не ссылается на `Foo.prototype`, как мы того хотим. Поэтому мы создаем *новый** объект, который имеет нужную ссылку, и отбрасываем исходный, неправильно связанный объект.

Примечание: Типичная ошибка — пытаться использовать следующие варианты, думая, что они тоже сработают, но это приводит к неожиданным результатам:

```

// работает не так, как вы ожидаете!
Bar.prototype = Foo.prototype;

// работает почти так, как нужно,
// но с побочными эффектами, которые возможно нежелательны :(
Bar.prototype = new Foo();

```

`Bar.prototype = Foo.prototype` не создает новый объект, на который ссылалось бы `Bar.prototype`. Вместо этого `Bar.prototype` становится еще одной ссылкой на `Foo.prototype`, и в результате `Bar` напрямую связывается с **тем же самым объектом**, что и `Foo: Foo.prototype`. Это значит, что когда вы начнете присваивать значения, например `Bar.prototype.myLabel = ...`, вы будете изменять **не отдельный объект**, а общий объект `Foo.prototype`, что повлияет на любые объекты, привязанные к `Foo.prototype`. Это наверняка не то, чего вы хотите. В противном случае вам вообще не нужен `Bar`, и стоит использовать только `Foo`, сделав код проще.

`Bar.prototype = new Foo()` **действительно** создает новый объект, корректно привязанный к `Foo.prototype`. Но для этого используется "вызов конструктора" `Foo(..)`. Если эта функция имеет какие-либо побочные эффекты (логирование, изменение состояния, регистрация в других объектах, **добавление свойств в `this`**, и т.д.), то эти побочные эффекты сработают во время привязывания (и возможно в отношении неправильного объекта!), а не только при создании конечных "потомков" `Bar`, как можно было бы ожидать.

Поэтому, для правильного привязывания нового объекта без побочных эффектов от вызова `Foo(..)` у нас остается лишь `Object.create(..)`. Небольшой недостаток состоит в том,

что нам приходится создавать новый объект и выбрасывать старый, вместо того чтобы модифицировать существующий стандартный объект.

Было бы здорово если бы существовал стандартный и надежный способ поменять привязку существующего объекта. До ES6 был нестандартный и не полностью кросбраузерный способ через свойство `.__proto__`, которое можно изменять. В ES6 добавлена вспомогательная утилита `Object.setPrototypeOf(..)`, которая проделывает нужный трюк стандартным и предсказуемым способом.

Сравните пред-ES6 и стандартизованный в ES6 способ привязки `Bar.prototype` к `Foo.prototype`:

```
// пред-ES6
// выбрасывает стандартный существующий `Bar.prototype`
Bar.prototype = Object.create( Foo.prototype );

// ES6+
// изменяет существующий `Bar.prototype`
Object.setPrototypeOf( Bar.prototype, Foo.prototype );
```

Если отбросить небольшой проигрыш в производительности (выбрасывание объекта, который позже удаляется сборщиком мусора), то способ с `Object.create(..)` немного короче и может даже читабельнее, чем подход ES6+. Хотя это всего лишь пустые разговоры о синтаксисе.

› Инспектируем связи между "классами"

Что если у вас есть объект `a` и вы хотите выяснить, какому объекту он делегирует?

Инспектирование экземпляра (объект в JS) с целью найти его предка (делегирующая связь в JS) в традиционных класс-ориентированных языках часто называют *интроспекцией* (или *рефлексией*).

Рассмотрим:

```
function Foo() {
  // ...
}

Foo.prototype.blah = ...;

var a = new Foo();
```

Как выполнить интроспекцию `a`, чтобы найти его "предка" (делегирующую связь)? Первый подход использует путаницу с "классами":

```
a instanceof Foo; // true
```

Оператор `instanceof` принимает в качестве операнда слева обычный объект, а в качестве операнда справа — функцию. `instanceof` отвечает на следующий вопрос: **присутствует ли**

где-либо в цепочке [[Prototype]] объекта аобъект, на который указывает Foo.prototype?

К сожалению, это значит, что вы можете получить сведения о "происхождении" некоторого объекта (а) только имея некоторую **функцию** (Foo с её ссылкой .prototype). Если у вас есть два произвольных объекта, например a и b, и вы хотите узнать, связаны ли сами эти **объекты** друг с другом через цепочку [[Prototype]], одного instanceof будет недостаточно.

Примечание: Если вы используете встроенную утилиту .bind(..) для создания жестко привязанной функции (см. главу 2), то у созданной функции не будет свойства .prototype. При использовании instanceof с такой функцией прозрачно подставляется .prototype целевой функции, из которой была создана жестко привязанная функция.

Использование функции с жесткой привязкой для "вызыва конструктора" крайне маловероятно, но если вы сделаете это, то она будет вести себя так, как если бы вы вызвали **целевую функцию**. Это значит, что вызов instanceof с жестко привязанной функцией также ведет себя в соответствии с оригинальной функцией.

Этот фрагмент кода показывает нелепость попыток рассуждать об отношениях между **двумя объектами** используя семантику "классов" и instanceof:

```
// вспомогательная утилита для проверки
// связан ли `o1` (через делегирование) с `o2`
function isRelatedTo(o1, o2) {
    function F(){}
    F.prototype = o2;
    return o1 instanceof F;
}

var a = {};
var b = Object.create( a );

isRelatedTo( b, a ); // true
```

Внутри isRelatedTo(..) мы временно используем функцию F, меняя значение её свойства .prototype на объект o2, а затем спрашиваем, является ли o1 "экземпляром" F. Ясно, что o1 *на самом деле* не унаследован от и даже не создан с помощью F, поэтому должно быть понятно, что подобные приемы бессмысленны и сбивают с толку. **Проблема сводится к несуразности семантики классов, навязываемой JavaScript**, что наглядно показано на примере косвенной семантики instanceof.

Второй подход к рефлексии [[Prototype]] более наглядный:

```
Foo.prototype.isPrototypeOf( a ); // true
```

Заметьте, что в этом случае нам неинтересна (и даже *не нужна*) Foo. Нам просто нужен **объект** (в данном случае произвольный объект с именем Foo.prototype) для сопоставления его с другим **объектом**. isPrototypeOf(..) отвечает на вопрос: **присутствует ли где-либо в цепочке [[Prototype]] объекта a объект Foo.prototype?**

Тот же вопрос, и в точности такой же ответ. Но во втором подходе нам не нужна **функция** (Foo) для косвенного обращения к её свойству .prototype.

Нам просто нужны два **объекта** для выявления связи между ними. Например:

```
// Просто: присутствует ли `b` где-либо
// в цепочке [[Prototype]] объекта `c`
b.isPrototypeOf( c );
```

Заметьте, что в этом подходе вообще не требуется функция ("класс"). Используются прямые ссылки на объекты b и c, чтобы выяснить нет ли между ними связи. Другими словами, наша утилита isRelatedTo(..) уже встроена в язык и называется isPrototypeOf(..).

Мы можем напрямую получить [[Prototype]] объекта. В ES5 появился стандартный способ сделать это:

```
Object.getPrototypeOf( a );
```

Здесь видно, что ссылка на объект является тем, что мы и ожидаем:

```
Object.getPrototypeOf( a ) === Foo.prototype; // true
```

В большинстве браузеров (но не во всех!) давно добавлена поддержка нестандартного альтернативного способа доступа к [[Prototype]]:

```
a.__proto__ === Foo.prototype; // true
```

Загадочное свойство __proto__ (стандартизовано лишь в ES6!) "магически" возвращает ссылку на внутреннее свойство [[Prototype]] объекта, что весьма полезно, если вы хотите напрямую проинспектировать (или даже обойти: __proto__.__proto__...) цепочку.

Аналогично рассмотренному ранее .constructor, свойство __proto__ отсутствует у инспектируемого объекта (а в нашем примере). На самом деле оно есть (и является неперечисляемым, см. главу 2) у встроенного Object.prototype, наряду с другими известными утилитами (.toString(), .isPrototypeOf(..), и т.д.).

Более того, __proto__ выглядит как свойство, но правильнее думать о нем как о геттере/сеттере (см. главу 3).

Грубо говоря, можно представить, что __proto__ реализовано так (см. главу 3 об определениях свойств объекта):

```
Object.defineProperty( Object.prototype, "__proto__", {
    get: function() {
        return Object.getPrototypeOf( this );
    },
    set: function(o) {
```

```
// setPrototypeOf(..) доступно начиная с ES6
Object.setPrototypeOf( this, o );
return o;
}
} );
```

Таким образом, когда мы обращаемся (получаем значение) к `a.__proto__` это похоже на вызов `a.__proto__()` (вызов функции геттера). В этом* вызове функции `this` указывает на `a`, несмотря на то что функция геттера находится в объекте `Object.prototype` (см. главу 2 о правилах привязки `this`), так что это равносильно `Object.getPrototypeOf(a)`.

Значение `__proto__` можно также изменять, например с помощью функции `Object.setPrototypeOf(..)` в ES6, как показано выше. Однако обычно **не следует изменять `[[Prototype]]` существующего объекта**.

Глубоко внутри некоторых фреймворков можно встретить сложные, продвинутые механизмы, позволяющие выполнять трюки наподобие "создания производного класса" от `Array`, но обычно подобные вещи не поощряются в повседневной практике, поскольку такой код гораздо труднее понимать и сопровождать.

Примечание: В ES6 добавлено ключевое слово `class`, с помощью которого можно делать вещи, напоминающие "создание производных классов" от встроенных объектов, таких как `Array`. Обсуждение синтаксиса `class` из ES6 см. в Приложении А.

В остальном же единственным исключением будет установка свойства `[[Prototype]]` стандартного объекта `.prototype` функции, чтобы оно ссылалось на какой-то другой объект (помимо `Object.prototype`). Это позволит избежать замены этого стандартного объекта новым объектом. В любой другой ситуации **лучше всего считать ссылку `[[Prototype]]` доступной только для чтения**, чтобы облегчить чтение вашего кода в будущем.

Примечание: В сообществе JavaScript неофициально закрепилось название двойного подчеркивания, особенно перед именами свойств (как `__proto__`): "dunder". Поэтому в мире JavaScript "крутые ребята" обычно произносят `__proto__` как "dunder proto".

Объектные ссылки

Как мы уже видели, механизм `[[Prototype]]` является внутренней ссылкой, существующей у объекта, который ссылается на другой объект.

Переход по этой ссылке выполняется (в основном) когда происходит обращение к свойству/методу первого объекта, и это свойство/метод отсутствует. В таком случае ссылка `[[Prototype]]` указывает движку, что свойство/метод нужно искать в связанном объекте. Если и в этом объекте ничего не находится, то происходит переход по его ссылке `[[Prototype]]`, и так далее. Эта последовательность ссылок между объектами образует то, что называется "цепочкой прототипов".

Создание ссылок

Мы подробно объяснили, почему механизм `[[Prototype]]` в JavaScript не похож на "классы", и увидели, что вместо этого создаются **ссылки** между подходящими объектами.

В чем смысл механизма `[[Prototype]]`? Почему многие JS разработчики прикладывают так много усилий (эммулируя классы), чтобы настроить эти ссылки?

Помните, как в начале этой главы мы сказали, что `Object.create(..)` станет нашим героям? Теперь вы готовы это увидеть.

```
var foo = {
    something: function() {
        console.log( "Скажи что-нибудь хорошее..." );
    }
};

var bar = Object.create( foo );

bar.something(); // Скажи что-нибудь хорошее...
```

`Object.create(..)` создает новый объект (`bar`), связанный с объектом, который мы указали (`foo`), и это дает нам всю мощь (делегирование) механизма `[[Prototype]]`, но без ненужных сложностей вроде функции `new`, выступающей в роли классов и вызовов конструктора, сбивающих с толку ссылок `.prototype` и `.constructor`, и прочих лишних вещей.

Примечание: `Object.create(null)` создает объект с пустой (или `null`) ссылкой `[[Prototype]]`, поэтому этот объект не сможет ничего делегировать. Поскольку у такого объекта нет цепочки прототипов, оператору `instanceof`(рассмотренному ранее) нечего проверять, и он всегда вернет `false`. Эти специальные объекты с пустым `[[Prototype]]` часто называют "словарями", поскольку они обычно используются исключительно для хранения данных в свойствах, потому что у них не может быть никаких побочных эффектов от делегируемых свойств/функций цепочки `[[Prototype]]`, и они являются абсолютно плоскими хранилищами данных.

Для создания продуманных связей между двумя объектами нам не нужны классы. Нам нужно **только лишь** связать объекты друг с другом для делегирования, и `Object.create(..)` дает нам эту связь без лишней возни с классами.

› Полифилл для `Object.create()`

`Object.create(..)` была добавлена в ES5. Вам может понадобиться поддержка пред-ES5 окружения (например, старые версии IE), поэтому давайте рассмотрим простенький **частичный** полифилл для `Object.create(..)`:

```
if (!Object.create) {
    Object.create = function(o) {
        function F(){}
        F.prototype = o;
        return new F();
    };
}
```

В этом полифилле используется временно создаваемая функция F, и её свойство .prototype переопределяется так, чтобы указывать на объект, с которым нужно создать связь. Затем мы используем new F(), чтобы создать новый объект, который будет привязан нужным нам образом.

Такой вариант использования Object.create(..) встречается в подавляющем большинстве случаев, поскольку эту часть можно заменить полифиллом. В ES5 стандартная функция Object.create(..) предоставляет дополнительную функциональность, которую нельзя заменить полифиллом в пред-ES5. Для полноты картины рассмотрим, в чем она заключается:

```
var anotherObject = {
  a: 2
};

var myObject = Object.create( anotherObject, {
  b: {
    enumerable: false,
    writable: true,
    configurable: false,
    value: 3
  },
  c: {
    enumerable: true,
    writable: false,
    configurable: false,
    value: 4
  }
} );

myObject.hasOwnProperty( "a" ); // false
myObject.hasOwnProperty( "b" ); // true
myObject.hasOwnProperty( "c" ); // true

myObject.a; // 2
myObject.b; // 3
myObject.c; // 4
```

Второй аргумент Object.create(..) указывает свойства, которые будут добавлены в создаваемый объект, объявляя *дескриптор* каждого нового свойства (см. главу 3). Поскольку полифилинг дескрипторов свойств в пред-ES5 невозможен, эту дополнительную функциональность Object.create(..) также невозможно реализовать в виде полифилла.

В большинстве случаев используется лишь та часть функциональности Object.create(..), которую можно заменить полифиллом, поэтому большинство разработчиков устраивает использование *частичного полифилла* в пред-ES5 окружениях.

Некоторые разработчики придерживаются более строгого подхода, считая, что можно использовать только *полные полифиллы*. Поскольку Object.create(..) — одна из тех утилит, что нельзя полностью заменить полифиллом, такой строгий подход предписывает, что если вы хотите использовать Object.create(..) в пред-ES5 окружении, то вместо полифилла

следует применить собственную утилиту, и вообще воздержаться от использования имени `Object.create`. Вместо этого можно определить свою собственную утилиту:

```
function createAndLinkObject(o) {
    function F(){}
    F.prototype = o;
    return new F();
}

var anotherObject = {
    a: 2
};

var myObject = createAndLinkObject( anotherObject );

myObject.a; // 2
```

Я не разделяю такой подход. Меня полностью устраивает показанный выше частичный полифилл `Object.create`(...) и его использование в коде даже в пред-ES5. Решайте сами, какой подход вам ближе.

Ссылки в роли запасных свойств?

Существует соблазн думать, что эти ссылки между объектами в основном предоставляют что-то вроде запасного варианта на случай "отсутствующих" свойств или методов. И хотя такой вывод допустим, я не считаю что это верный способ размышлений о [[Prototype]].

Рассмотрим:

```
var anotherObject = {
    cool: function() {
        console.log( " круто!" );
    }
};

var myObject = Object.create( anotherObject );

myObject.cool(); // " круто!"
```

Этот код работает благодаря [[Prototype]], но если вы написали его так, что `anotherObject` играет роль запасного варианта на случай если `myObject` не сможет обработать обращение к некоторому свойству/методу, то вероятно ваше ПО будет содержать больше "магии" и будет сложнее для понимания и сопровождения.

Я не хочу сказать, что такой подход является в корне неверным шаблоном проектирования, но он нехарактерен для JS. Если вы используете его, возможно вам стоит сделать шаг назад и подумать, является ли такое решение уместным и разумным.

Примечание: В ES6 добавлена продвинутая функциональность, называемая `Proxy`, с помощью которой можно реализовать что-то наподобие поведения "отсутствующих

методов". Proxy выходит за рамки этой книги, но будет подробно рассмотрена в одной из следующих книг серии "Вы не знаете JS".

Не упустите одну важную, но едва уловимую мысль.

Явно проектируя ПО таким образом, что разработчик может вызвать, к примеру, `myObject.cool()`, и это будет работать даже при отсутствии метода `cool()` у `myObject`, вы добавляете немного "магии" в дизайн вашего API, что может в будущем преподнести сюрприз другим разработчикам, которые будут поддерживать ваш код.

Но вы можете спроектировать API и без подобной "магии", не отказываясь при этом от преимуществ ссылки `[[Prototype]]`.

```
var anotherObject = {
    cool: function() {
        console.log( " круто! " );
    }
};

var myObject = Object.create( anotherObject );

myObject.doCool = function() {
    this.cool(); // внутреннее делегирование!
};

myObject.doCool(); // " круто!"
```

Здесь мы вызываем `myObject.doCool()` — метод, который *действительно есть* у объекта `myObject`, делая наш API более явным (менее "магическим"). Внутри наша реализация следует **шаблону делегирования** (см. главу 6), используя делегирование `[[Prototype]]` к `anotherObject.cool()`.

Другими словами, делегирование как правило преподносит меньше сюрпризов, если оно является частью внутренней реализации, а не выставлено наружу в дизайне API. Мы изучим **делегирование** в мельчайших подробностях в следующей главе.

› Обзор

При попытке обратиться к несуществующему свойству объекта внутренняя ссылка `[[Prototype]]` этого объекта задает дальнейшее направление поиска для операции `[[Get]]` (см. главу 3). Этот каскад ссылок от объекта к объекту образует "цепочку прототипов" (чем то похожую на цепочку вложенных областей видимости) для обхода при разрешении свойства.

У обычных объектов есть встроенный объект `Object.prototype` на конце цепочки прототипов (похоже на глобальную область видимости при поиске по цепочке областей видимости), где процесс разрешения свойства остановится, если свойство не будет найдено в предыдущих звеньях цепочки. У этого объекта есть утилиты `toString()`, `valueOf()` и несколько других, благодаря чему все объекты в языке имеют доступ к ним.

Наиболее популярный способ связать два объекта друг с другом — использовать ключевое слово `new` с вызовом функции, что помимо четырех шагов (см. главу 2) создаст новый объект, привязанный к другому объекту.

Этим "другим объектом" является объект, на который указывает свойство `.prototype` функции, вызванной с `new`. Функции, вызываемые с `new`, часто называют "конструкторами", несмотря на то что они не создают экземпляры классов, как это делают *конструкторы* в традиционных класс-ориентированных языках.

Хотя эти механизмы JavaScript могут напоминать "создание экземпляров классов" и "наследование классов" из традиционных класс-ориентированных языков, ключевое отличие в том, что в JavaScript не создаются копии. Вместо этого объекты связываются друг с другом через внутреннюю цепочку `[[Prototype]]`.

По множеству причин, среди которых не последнюю роль играет терминологический прецедент, "наследование" (и "прототипное наследование") и все остальные ОО-термины не имеют смысла, учитывая то как *на самом деле* работает JavaScript.

Более подходящим термином является "делегирование", поскольку эти связи являются *не копиями*, а делегирующими *ссылками*.

Глава 6: Делегирование поведения

В главе 5 мы подробно изучили механизм `[[Prototype]]` и показали, почему его нельзя корректно описать в терминах "класс" или "наследование" (несмотря на бесчисленные попытки на протяжении почти двух десятилетий). Нам пришлось прорыться не только через многословный синтаксис (`.prototype`, захламляющий код), но и через всевозможные ловушки (такие как непредвиденное разрешение `.constructor` или уродливый псевдополиморфный синтаксис). Мы также рассмотрели различные варианты использования "примесей", которые часто используются, чтобы сгладить эти острые углы.

Возникает закономерный вопрос: почему так сложно делать такие простые вещи? Теперь, когда мы приоткрыли завесу и увидели, насколько грязно все устроено внутри, неудивительно, что большинство JS разработчиков никогда не погружаются так глубоко, поручая эту работу библиотеке "классов".

Я надеюсь, что вы не собираетесь просто обойти все эти детали, поручив их "черному ящику". Так что давайте разберемся, как мы *могли и должны были бы* думать о механизме `[[Prototype]]` в JS, используя гораздо более простой и прямой путь, чем вся эта путаница с классами.

Как вы уже знаете из Главы 5, механизм `[[Prototype]]` — это внутренняя ссылка, которая существует в одном объекте и ссылается на другой объект.

Эта ссылка используется при обращении к несуществующему свойству/методу первого объекта. В таком случае ссылка `[[Prototype]]` говорит движку, что свойство/метод нужно искать в связанном объекте. В свою очередь, если поиск в этом объекте завершается неудачно, то происходит переход уже по его ссылке `[[Prototype]]` и так далее. Эта последовательность ссылок между объектами образует так называемую "цепочку прототипов".

Другими словами, реальный механизм, важнейшая часть функциональности, доступной нам в JavaScript — это по сути **объекты, связанные с другими объектами**.

Данное наблюдение является фундаментальным и критически важным для понимания мотивов и подходов, описанных далее в этой главе!

› Towards Delegation-Oriented Design

Чтобы использовать `[[Prototype]]` наиболее правильным способом, необходимо осознавать, что этот шаблон проектирования фундаментально отличается от классов (см. главу 4).

Примечание: Некоторые принципы класс-ориентированного проектирования остаются крайне актуальными, так что не отбрасывайте все, что вы знаете (а всего лишь большую часть!). Например, *инкапсулирование* — весьма мощный инструмент, совместимый с делегированием (хотя такое сочетание встречается редко).

Нам нужно изменить наш способ мышления с шаблона проектирования "класс/наследование" на шаблон проектирования "делегирование поведения". Если большую часть

вашего обучения или карьеры в программировании вы имели дело с классами, это может показаться некомфортным или неестественным. Попробуйте проделать это умственное упражнение несколько раз, чтобы привыкнуть к такому совершенно иному способу мышления.

Сначала я покажу вам некоторые теоретические упражнения, а затем мы посмотрим на более конкретный пример, который вы сможете использовать на практике в вашем коде.

Теория классов

Предположим, что у нас есть несколько похожих задач ("XYZ", "ABC", etc), которые мы хотим смоделировать в нашем ПО.

При использовании классов проектирование происходит так: определяем общий родительский (базовый) класс Task, в котором задается поведение всех "похожих" задач. Затем определяем дочерние классы XYZ и ABC, которые наследуют от Task и добавляют уточненное поведение для выполнения собственных задач.

Важно отметить, что шаблон проектирования классов диктует нам для получения максимальной выгоды от наследования использовать переопределение методов (и полиморфизм). В этом случае мы переопределяем определение некоторого общего метода Task в XYZ, возможно даже используя super для вызова базовой версии метода, добавляя к нему новое поведение. **Вероятно вы найдете довольно много мест**, где можно "абстрагировать" общее поведение в родительский класс, и уточнить (переопределить) его в дочерних классах.

Вот примерный псевдокод для такого сценария:

```
class Task {  
    id;  
  
    // конструктор `Task()`  
    Task(ID) { id = ID; }  
    outputTask() { output(id); }  
}  
  
class XYZ inherits Task {  
    label;  
  
    // конструктор `XYZ()`  
    XYZ(ID,Label) { super( ID ); label = Label; }  
    outputTask() { super(); output( label ); }  
}  
  
class ABC inherits Task {  
    // ...  
}
```

Теперь вы можете создать одну или более **копий** дочернего класса XYZ, и использовать эти экземпляры для выполнения задачи "XYZ". Эти экземпляры **копируют** как общее поведение из Task, так и уточненное поведение из xyz. Аналогично и экземпляры класса abc будут иметь

копии поведения Task и уточненного поведения abc. Обычно после создания вы взаимодействуете только с этими экземплярами (но не с классами), поскольку у каждого экземпляра есть копия всего поведения, которое необходимо для выполнения задачи.

› Теория делегирования

А теперь давайте поразмышляем о той же предметной области, но с использованием *делегирования поведения* вместо классов.

Сначала определяется **объект** (не класс и не function, что бы ни говорили вам большинство JS разработчиков) по имени Task с конкретным поведением, включающим в себя вспомогательные методы, которыми могут пользоваться различные задачи (читай *делегировать!*). Затем для каждой задачи ("XYZ", "ABC") вы определяете **объект** с данными/поведением, специфичными для данной задачи. Вы **связываете** специфические объекты задач со вспомогательным объектом Task, позволяя им делегировать ему в случае необходимости.

В сущности, для выполнения задачи "XYZ" нам необходимо поведение двух объектов одного уровня (XYZ и Task). Но вместо композиции через копирование классов мы можем оставить их в виде отдельных объектов, и разрешить объекту XYZ **делегировать** объекту Task, когда это необходимо.

Вот простой пример кода, показывающий как этого добиться:

```
var Task = {
    setID: function(ID) { this.id = ID; },
    outputID: function() { console.log( this.id ); }
};

// `XYZ` делегирует `Task`
var XYZ = Object.create( Task );

XYZ.prepareTask = function(ID,Label) {
    this.setID( ID );
    this.label = Label;
};

XYZ.outputTaskDetails = function() {
    this.outputID();
    console.log( this.label );
};

// ABC = Object.create( Task );
// ABC ... = ...
```

В этом примере Task и XYZ не являются классами (или функциями), это **просто объекты**. С помощью Object.create(..) объект XYZ делегирует объекту Task через ссылку [[Prototype]] (см. главу 5).

По аналогии с класс-ориентированностью (или, ОО — объектно-ориентированный), я назвал этот стиль кода "**OLOO**"(objects-linked-to-other-objects — "объекты, связанные с другими

объектами"). Все, что нас *действительно* интересует — это тот факт, что объект xyz делегирует объекту Task (как и объект abc).

В JavaScript механизм `[[Prototype]]` связывает **объекты** с другими **объектами**. Нет никаких абстрактных механизмов наподобие "классов", как бы вы ни пытались убедить себя в обратном. Это как грести на каноэ вверх по реке: вы *можете* это сделать, но *выбираете* путь против естественного течения, так что вам очевидно будет труднее добраться в нужное место.

Вот некоторые другие отличия стиля OOO:

1. Оба члена данных `id` и `label` из предыдущего примера с классами являются здесь свойствами данных непосредственно xyz (ни одного из них нет в Task). Как правило в случае делегирования через `[[Prototype]]`, вы хотите, чтобы состояние хранилось в **делегирующих объектах** (xyz, abc), а не в делегате (Task).
2. При использовании шаблона проектирования классов мы специально назвали `outputTask` одинаково как в родителе (`Task`), так и в потомке (`xyz`), чтобы воспользоваться переопределением (полиморфизм). В случае делегирования поведения мы делаем ровно наоборот: **при любой возможности избегаем одинаковых имен** на разных уровнях цепочки `[[Prototype]]` (это называется затенением — см. главу 5), поскольку коллизии имен вынуждают использовать ужасный/хрупкий код для устранения неоднозначности ссылок (см. главу 4), а мы хотим избежать этого.
3. `this.setID(ID);` внутри метода объекта xyz сначала ищет `setID(..)` в xyz, но поскольку метода с таким именем нет в xyz, **делегирование `[[Prototype]]`** означает, что можно пройти по ссылке на Task, чтобы найти там `setID(..)`, что и происходит. Более того, благодаря неявным правилам привязки `this` (см. главу 2), при выполнении `setID(..)`, хотя этот метод и был найден в Task, `this` для данного вызова функции — это xyz, как мы того и желали. То же самое происходит и с `this.outputID()` чуть дальше в листинге кода.

Другими словами, методы общего назначения, существующие в Task, доступны нам при взаимодействии с xyz, потому что xyz может делегировать Task.

Делегирование поведения означает: пусть у одного объекта (xyz) будет делегирование (к Task) для обращения к свойству или методу, отсутствующему в объекте (xyz).

Это *чрезвычайно мощный* шаблон проектирования, сильно отличающийся от идеи родительских и дочерних классов, наследования, полиморфизма, и т.п. Вместо того чтобы мысленно выстраивать вертикальную иерархию объектов от Родителей к Потомкам, представьте себе равноправные объекты одного уровня, между которыми в любом направлении могут идти делегирующие ссылки.

Примечание: Правильнее использовать делегирование как внутреннюю деталь реализации, а не выставлять его наружу в дизайне API. В нашем дизайне API в примере выше мы не

подталкиваем разработчиков использовать `XYZ.setID()` (хотя могли бы это сделать!). Мы как бы прячем делегирование как внутреннюю деталь нашего API, где `XYZ.prepareTask(..)` делегирует к `Task.setID(..)`. Подробнее см. главу 5, раздел "Ссылки в роли запасных свойств".

› Взаимное делегирование (запрещено)

Нельзя создавать цикл, где между двумя или более объектами есть взаимное (дву направленное) делегирование. Если вы создадите `B`, связанный с `A`, а затем попытаетесь связать `A` с `B`, то получите ошибку.

Жаль, что это запрещено (не то чтобы ужасно, но слегка раздражает). Если бы вы обратились к свойству/методу, которого нет ни у одного из объектов, это привело бы к бесконечной рекурсии в цикле `[[Prototype]]`. Но если бы все ссылки были на месте, тогда мог делегировать `A` и наоборот, и это могло бы сработать. Это позволило бы использовать любой из объектов для делегирования другому. Есть несколько частных случаев, где это было бы полезным.

Но это запрещено, потому что разработчики конкретных реализаций движка обнаружили, что с точки зрения производительности выгоднее проверить (и отклонить!) наличие бесконечных циклических ссылок один раз во время установки, чем выполнять эту проверку каждый раз при обращении к свойству объекта.

› Отладка

Рассмотрим вкратце один тонкий момент, который иногда сбивает с толку разработчиков. В целом спецификация JS не контролирует то, в каком виде конкретные значения и структуры отображаются в консоли разработчика в браузере. Поэтому каждый браузер/движок интерпретирует подобные вещи по-своему, и эта интерпретация может различаться. В частности поведение, которое мы сейчас рассмотрим, встречается только в Chrome Developer Tools.

Посмотрите на этот "традиционный" стиль "конструктора классов" в JS коде, и то, что отображается в консоли Chrome Developer Tools:

```
function Foo() {}

var a1 = new Foo();

a1; // Foo {}
```

Обратите внимание на последнюю строку кода: в результате вычисления выражения `a1` выводится `Foo {}`. Если запустить этот код в Firefox, то скорее всего мы увидим `Object {}`. Почему такая разница, и что означают эти значения в консоли?

Chrome по сути говорит нам, что `{} — это пустой объект, который был создан функцией с именем 'Foo'"`. Firefox же говорит, что `{} — это пустой объект, созданный на основе Object"`. Маленькое отличие состоит в том, что Chrome отслеживает в виде *внутреннего* свойства имя реальной функции, создавшей объект, а другие браузеры такую информацию не отслеживают.

Есть соблазн попытаться объяснить это с помощью механизмов JavaScript:

```
function Foo() {}

var a1 = new Foo();

a1.constructor; // Foo(){}
a1.constructor.name; // "Foo"
```

Получается, что Chrome выводит "Foo", всего-навсего проверяя свойство `.constructor.name` объекта? И "да", и "нет".

Рассмотрим код:

```
function Foo() {}

var a1 = new Foo();

Foo.prototype.constructor = function Gotcha(){}

a1.constructor; // Gotcha(){}
a1.constructor.name; // "Gotcha"

a1; // Foo {}
```

Несмотря на то, что мы изменили `a1.constructor.name` на другое значение ("Gotcha"), консоль Chrome по-прежнему использует имя "Foo".

Итак, получается, что ответ на предыдущий вопрос (используется ли `.constructor.name`?) — **нет**, информация отслеживается где-то в другом месте, внутри движка.

Но не торопитесь! Давайте посмотрим, как это работает при использовании стиля OOO:

```
var Foo = {};

var a1 = Object.create( Foo );

a1; // Object {}

Object.defineProperty( Foo, "constructor", {
    enumerable: false,
    value: function Gotcha(){}
});

a1; // Gotcha {}
```

Ага! Попались! Здесь консоль Chrome нашла и использует `.constructor.name`. На самом деле, на момент написания этой книги данное поведение было идентифицировано как баг в Chrome, и сейчас, когда вы её читаете, баг скорее всего исправлен. Поэтому в вашем браузере может выдаваться корректный результат `a1; // Object {}`.

Если не обращать внимания на этот баг, то внутреннее отслеживание "имени конструктора" (по-видимому, только в целях отладки в консоли) в Chrome является собственным поведением Chrome, выходящим за рамки спецификации.

Если вы не используете "конструктор" для создания объектов, как того и требует OOO стиль кодирования в этой главе, тогда Chrome *не* будет отслеживать внутреннее "имя конструктора" для этих объектов, и они будут отображаться в консоли как "Object {}", то есть, "объекты, созданные из Object()".

Не думайте, что это является недостатком OOO стиля. Когда вы используете шаблон проектирования на основе OOO и делегирования поведения, совершенно неважно, кто "создал" объект (то есть, *какая функция* была вызвана с new?). Отслеживание "имени конструктора" внутри Chrome полезно только если вы полностью пишете код "в стиле классов", но совершенно неактуально при использовании OOO делегирования.

Сравнение мысленных моделей

Теперь, когда вы видите как минимум теоретическую разницу между шаблонами проектирования "класс" и "делегирование", давайте посмотрим на то, как эти шаблоны влияют на мысленные модели, которые мы используем, рассуждая о коде.

Мы рассмотрим абстрактный код ("Foo", "Bar"), и сравним два способа его реализации (OO против OOO). Первый фрагмент кода использует классический ("прототипный") ОО стиль:

```
function Foo(who) {
    this.me = who;
}
Foo.prototype.identify = function() {
    return "I am " + this.me;
};

function Bar(who) {
    Foo.call( this, who );
}
Bar.prototype = Object.create( Foo.prototype );

Bar.prototype.speak = function() {
    alert( "Hello, " + this.identify() + "." );
};

var b1 = new Bar( "b1" );
var b2 = new Bar( "b2" );

b1.speak();
b2.speak();
```

Родительский класс Foo наследуется дочерним классом Bar, после чего создаются два экземпляра этого класса b1 и b2. В результате b1 делегирует Bar.prototype, который делегирует Foo.prototype. Все выглядит довольно знакомо, ничего особенного.

Теперь давайте реализуем **ту же самую функциональность**, используя код в стиле OOO:

```
var Foo = {
    init: function(who) {
        this.me = who;
    },
    identify: function() {
        return "I am " + this.me;
    }
};

var Bar = Object.create(Foo);

Bar.speak = function() {
    alert("Hello, " + this.identify() + ".");
};

var b1 = Object.create(Bar);
b1.init("b1");
var b2 = Object.create(Bar);
b2.init("b2");

b1.speak();
b2.speak();
```

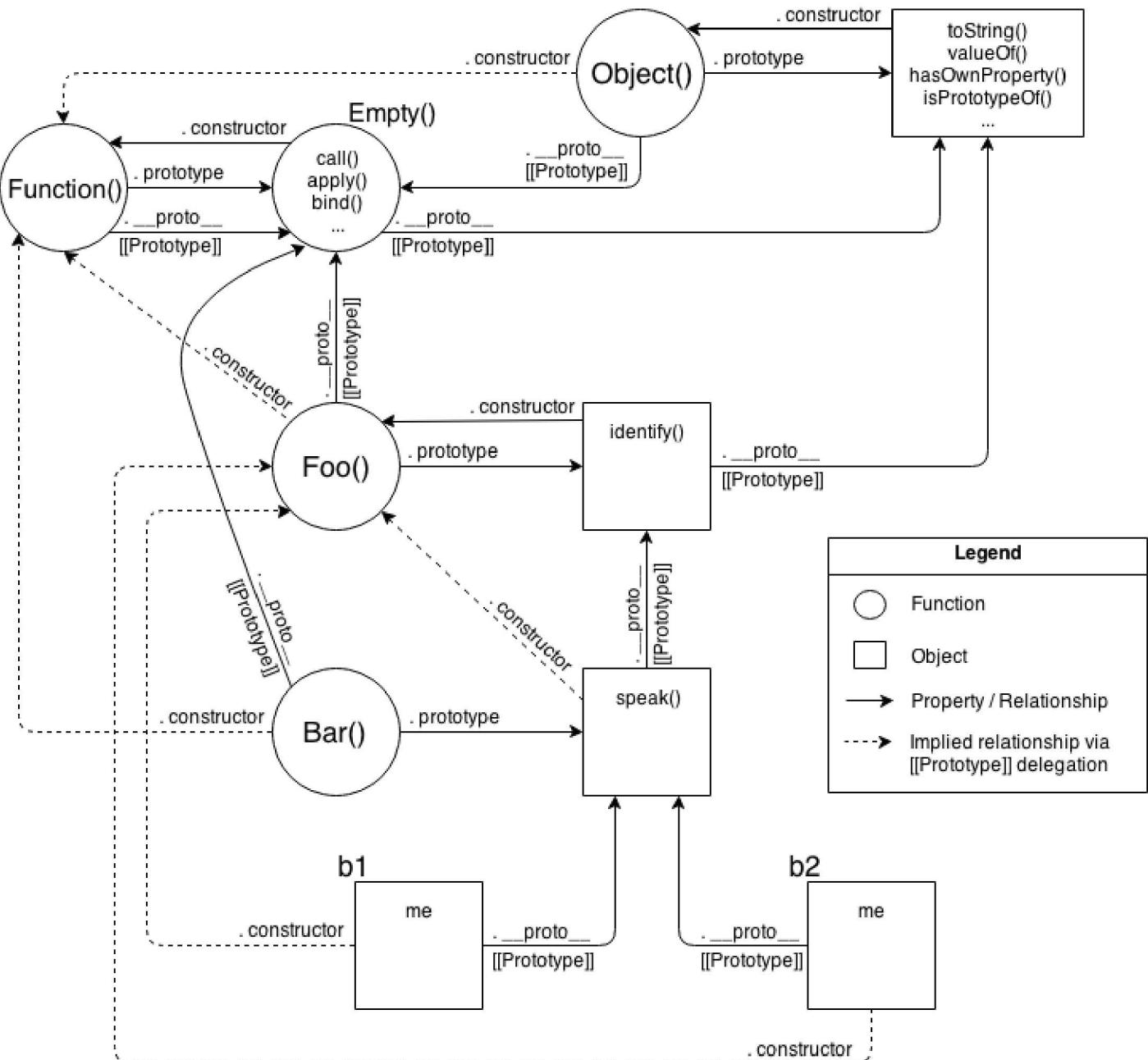
Мы используем преимущество делегирования [[Prototype]] от b1 к Bar, и от Bar к Foo, аналогично тому, как сделали это в предыдущем примере с b1, Bar.prototype, и Foo.prototype. У нас по-прежнему есть те же самые 3 объекта, связанные вместе.

Но важно то, что мы значительно упростили все остальное, потому что теперь у нас просто есть **объекты**, связанные друг с другом, без всех этих ненужных вещей, которые выглядят (но не ведут себя) как классы, с конструкторами, прототипами и вызовами new.

Спросите себя: если я могу получить с OOO точно такую же функциональность, что и с "классами", но OOO проще и понятнее, **может быть OOO лучше?**

Давайте рассмотрим мысленные модели, связанные с двумя этими примерами.

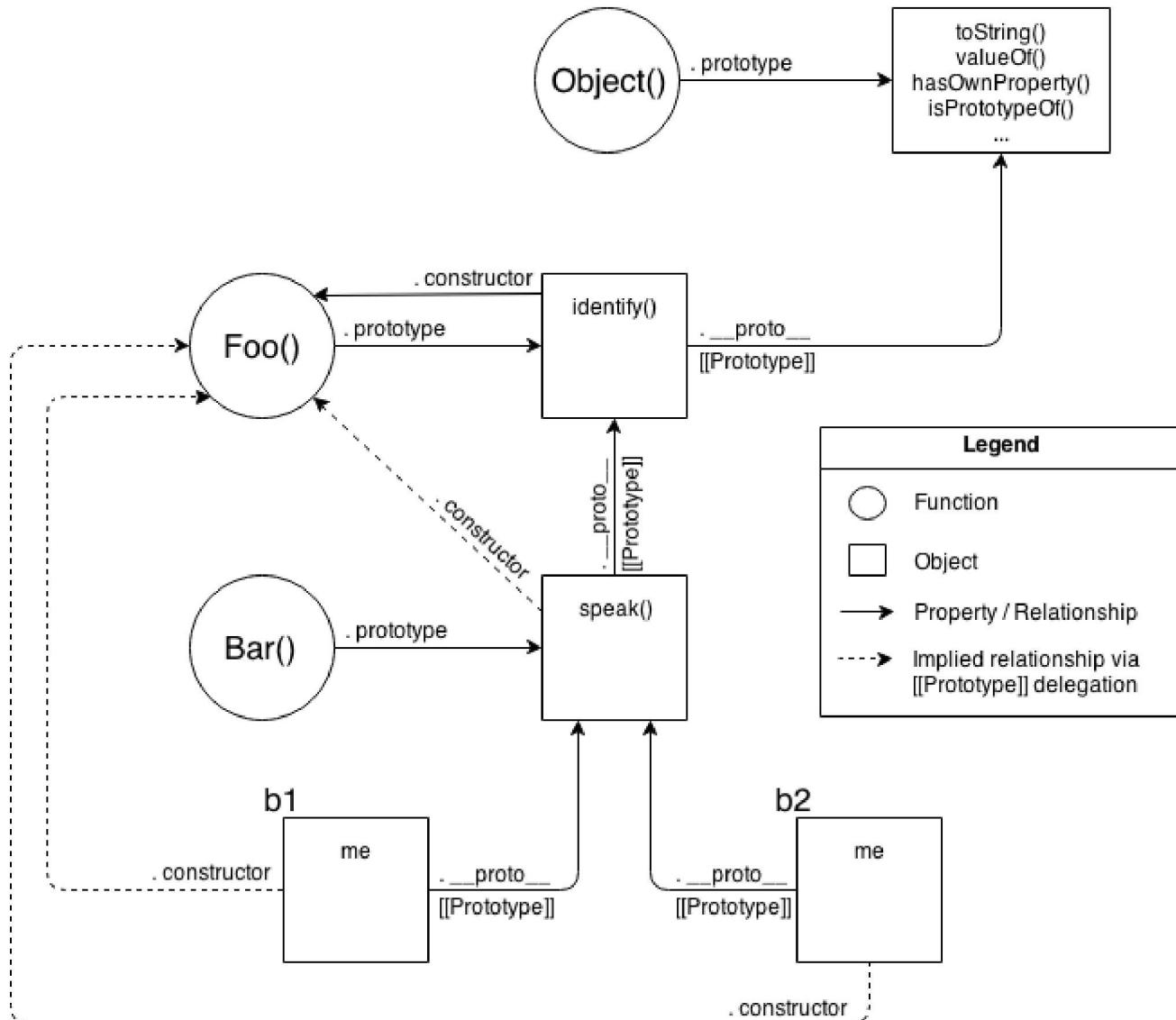
Пример с классами предполагает следующую мысленную модель сущностей и взаимосвязей между ними:



На самом деле, это немного нечестно, потому что здесь показано множество дополнительных нюансов, которые вы *строго говоря* не должны постоянно держать в голове (хотя вам *надо* понимать их!). С одной стороны, это довольно сложная последовательность взаимосвязей. Но с другой стороны, если вы внимательно изучите эти стрелки со связями, то поймете, что механизмы JS обладают **потрясающей внутренней целостностью и непротиворечивостью**.

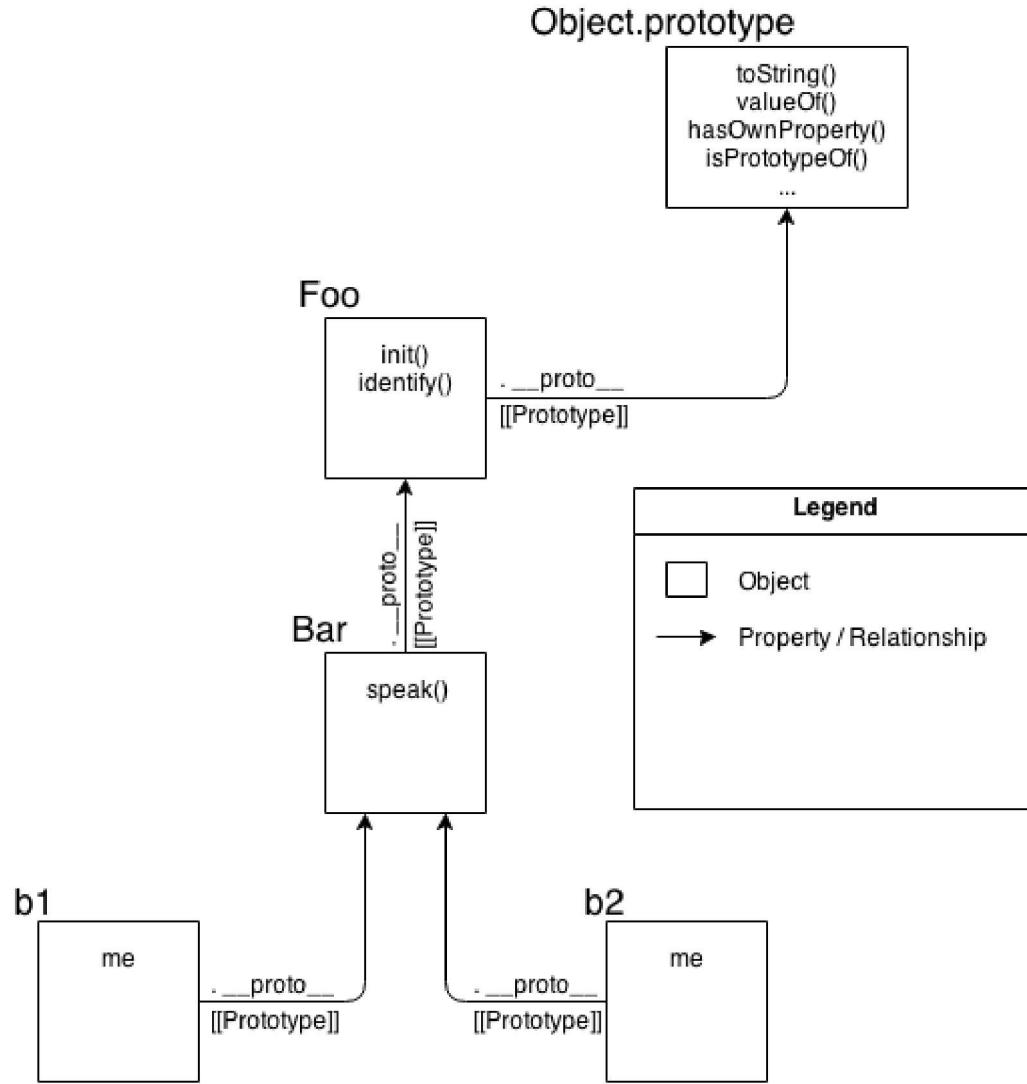
Например, функции в JS могут обращаться к `call(..)`, `apply(..)` и `bind(..)` (см. главу 2), поскольку сами по себе являются объектами, и у них есть ссылка `[[Prototype]]` на объект `Function.prototype`. В этом объекте определены стандартные методы, которым может делегировать любая функция-объект. JS может делать такие вещи, *и вы тоже можете!*

Хорошо, давайте теперь посмотрим на слегка упрощенную версию этой диаграммы, чтобы сделать наше сравнение чуть более "честным". Здесь показаны лишь **ключевые** сущности и взаимосвязи.



По-прежнему довольно сложно, не так ли? Пунктирными линиями обозначены неявные взаимосвязи, когда вы установили "наследование" между `Foo.prototype` и `Bar.prototype`, но пока еще не исправили ссылку на **отсутствующее** свойство `.constructor` (см. раздел "И снова о конструкторе" в главе 5). Даже без этих пунктирных линий вам придется мысленно проделывать очень много работы каждый раз, когда вы имеете дело с объектными.

А теперь давайте посмотрим на мысленную модель для кода в OOO-стиле:



Из этого сравнения очевидно, что в OOO-стиле вам нужно учитывать гораздо меньшее количество нюансов, поскольку в OOO принимается за аксиому тот факт, что нас интересуют только **объекты, связанные с другими объектами**.

Весь остальной "классовый" хлам — запутанный и сложный способ для получения такого же конечного результата. Уберите его, и вещи станут гораздо проще (без потери какой-либо функциональности).

› Классы против объектов

Мы только что провели теоретические рассуждения и сравнили мысленные модели "классов" и "делегирования поведения". А теперь давайте посмотрим более реальные примеры кода, чтобы увидеть как на деле применять эти идеи.

Сначала мы рассмотрим типичный сценарий фронтенд-разработки: создание UI-виджетов (кнопки, раскрывающиеся списки, и т.п.).

› "Классы" виджетов

Если вы привыкли использовать шаблон проектирования ОО, то скорее всего сразу же представите себе предметную область в виде родительского класса (например, `Widget`) с

базовым поведением виджета и дочерних производных классов для виджетов конкретного типа (например, Button).

Примечание: Для работы с DOM и CSS мы используем jQuery, поскольку в данном обсуждении нас не интересуют подобные детали. В приведенном ниже коде выбор конкретного JS фреймворка (jQuery, Dojo, YUI, и т.п.) для решения рутинных задач не имеет никакого значения.

Давайте посмотрим, как бы мы могли реализовать архитектуру "классов" на чистом JS, без каких-либо вспомогательных библиотек "классов" или синтаксиса:

```
// Родительский класс
function Widget(width,height) {
    this.width = width || 50;
    this.height = height || 50;
    this.$elem = null;
}

Widget.prototype.render = function($where){
    if (this.$elem) {
        this.$elem.css( {
            width: this.width + "px",
            height: this.height + "px"
        }).appendTo( $where );
    }
};

// Дочерний класс
function Button(width,height,label) {
    // вызов конструктора "super"
    Widget.call( this, width, height );
    this.label = label || "Default";

    this.$elem = $( "<button>" ).text( this.label );
}

// `Button` "наследует" от `Widget`
Button.prototype = Object.create( Widget.prototype );

// переопределяем базовый "унаследованный" `render(..)`
Button.prototype.render = function($where) {
    // вызов "super"
    Widget.prototype.render.call( this, $where );
    this.$elem.click( this.onClick.bind( this ) );
};

Button.prototype.onClick = function(evt) {
    console.log( "Button '" + this.label + "' clicked!" );
};

$( document ).ready( function(){
    var $body = $( document.body );
    var btn1 = new Button( 125, 30, "Hello" );
}
```

```
var btn2 = new Button( 150, 40, "World" );

btn1.render( $body );
btn2.render( $body );
} );
```

Шаблон проектирования ОО предписывает нам объявить базовый метод `render(..)` в родительском классе, и переопределить его в дочернем классе, но не заменять его полностью, а дополнить базовую функциональность поведением, характерным для кнопки.

Обратите внимание на уродливый явный псевдополиморфизм ссылок `Widget.call` и `Widget.prototype.render.call` для имитации вызова "super" из методов дочернего "класса". Фу, гадость!

› Синтаксический сахар ES6: `class`

Мы подробно рассмотрим синтаксический сахар `class` в ES6 в Приложении А. Ну а пока давайте узнаем, как мы могли бы реализовать тот же самый код с помощью `class`:

```
class Widget {
    constructor(width,height) {
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    }
    render($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            }).appendTo( $where );
        }
    }
}

class Button extends Widget {
    constructor(width,height,label) {
        super( width, height );
        this.label = label || "Default";
        this.$elem = $( "<button>" ).text( this.label );
    }
    render($where) {
        super.render( $where );
        this.$elem.click( this.onClick.bind( this ) );
    }
    onClick(evt) {
        console.log( "Button '" + this.label + "' clicked!" );
    }
}

$( document ).ready( function(){
    var $body = $( document.body );
```

```

var btn1 = new Button( 125, 30, "Hello" );
var btn2 = new Button( 150, 40, "World" );

btn1.render( $body );
btn2.render( $body );
} );

```

Несомненно, `class` в ES6 делает предыдущий классический код менее ужасным. В частности, довольно приятно наличие `super(..)` (хотя если копнуть поглубже, все не так красиво!).

Несмотря на улучшение синтаксиса, **это не настоящие классы**, поскольку они по-прежнему работают поверх механизма `[[Prototype]]`. Им присущи те же самые концептуальные несостыковки, рассмотренные нами в 4 и 5 главах, и в начале этой главы. В Приложении А мы подробно изучим синтаксис `class` в ES6 и последствия его применения. Мы увидим, почему устранение проблем с синтаксисом не избавляет нас от путаницы с классами в JS, хотя и преподносится как решение.

Неважно, используете ли вы классический прототипный синтаксис или новый синтаксический сахар ES6, вы по-прежнему моделируете предметную область с помощью "классов". И как показано в нескольких предыдущих главах, такой выбор в JavaScript сулит вам дополнительные проблемы и концептуальные трудности.

› Делегирование объектов виджетов

Вот более простая реализация нашего примера с `Widget` / `Button`, использующая **делегирование в стиле OOO**:

```

var Widget = {
    init: function(width,height){
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    },
    insert: function($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            }).appendTo( $where );
        }
    }
};

var Button = Object.create( Widget );

Button.setup = function(width,height,label){
    // делегированный вызов
    this.init( width, height );
    this.label = label || "Default";

    this.$elem = $( "<button>" ).text( this.label );
};

```

```

Button.build = function($where) {
    // делегированный вызов
    this.insert( $where );
    this.$elem.click( this.onClick.bind( this ) );
};

Button.onClick = function(evt) {
    console.log( "Button '" + this.label + "' clicked!" );
};

$( document ).ready( function(){
    var $body = $( document.body );

    var btn1 = Object.create( Button );
    btn1.setup( 125, 30, "Hello" );

    var btn2 = Object.create( Button );
    btn2.setup( 150, 40, "World" );

    btn1.build( $body );
    btn2.build( $body );
} );

```

Применяя OOO-стиль, мы не думаем о Widget и Button как о родительском и дочернем классах. Вместо этого, Widget — это просто объект, некий набор утилит, которым может делегировать любой конкретный тип виджета, а Button — это тоже самостоятельный объект (с делегирующей ссылкой на Widget, разумеется!).

Мы не используем в обоих объектах одно и то же имя метода render(..), как то предписывается шаблоном проектирования классов. Вместо этого мы выбрали разные имена (insert(..) и build(..)), которые более точно описывают решаемую каждым классом задачу. Инициализирующие методы названы init(..) и setup(..), соответственно, по тем же причинам.

Этот шаблон проектирования с использованием делегирования не только предлагает различающиеся и более содержательные имена (вместо одинаковых и более общих), но и избавляет нас от некрасивых явных псевдополиморфных вызовов (Widget.call и Widget.prototype.render.call), заменяя их на простые, относительные делегирующие вызовы this.init(..) и this.insert(..).

Из синтаксиса исчезли конструкторы, .prototype и new, поскольку на самом деле для нас они бесполезны.

Если вы были внимательны, то могли заметить, что вместо одного вызова (var btn1 = new Button(..)) у нас теперь два (var btn1 = Object.create(Button) и btn1.setup(..)). Поначалу это может показаться недостатком (больше кода).

Однако даже это является преимуществом кодирования в OOO-стиле по сравнению с классическим прототипным кодом. Почему?

Конструкторы классов "вынуждают" вас выполнять создание и инициализацию за один шаг (по крайней мере, это настоятельно рекомендуется). Однако во многих случаях нужна

большая гибкость и возможность выполнения этих этапов отдельно друг от друга (что и происходит в OOO!).

Допустим, в начале программы вы создаете все сущности и помещаете их в пул, но прежде чем извлечь их из этого пула и использовать, нужно дождаться, пока они не будут инициализированы. В примере выше оба вызова находятся рядом друг с другом, но разумеется их можно выполнять в совершенно разное время и в разных местах кода, если это необходимо.

OOO обеспечивает лучшую поддержку принципа разделения ответственности, поскольку создание и инициализацию необязательно объединять в одну операцию.

› Более простой дизайн

Помимо того, что OOO обеспечивает нарочито более простой (и гибкий!) код, делегирование поведения может упростить архитектуру кода. Давайте рассмотрим последний пример, показывающий, как OOO в целом упрощает дизайн.

В нашем примере будут два объекта-контроллера, один из которых обрабатывает форму входа на веб-странице, а другой отвечает за аутентификацию на сервере.

Нам понадобится вспомогательная утилита для Ajax-взаимодействия с сервером. Мы используем jQuery (хотя подойдет любой фреймворк), поскольку она не только выполняет за нас Ajax-запрос, но и возвращает в ответ promise (обещание), так что мы можем прослушивать ответ в вызывающем коде с помощью .then(..).

Примечание: Мы рассмотрим обещания (promises) в одной из будущих книг серии "Вы не знаете JS".

Придерживаясь типичного шаблона проектирования классов, мы разобьем задачу и вынесем базовую функциональность в класс Controller, а затем создадим два дочерних класса, LoginController и AuthController, унаследованных от Controller и уточняющих базовое поведение.

```
// Родительский класс
function Controller() {
    this.errors = [];
}
Controller.prototype.showDialog = function(title, msg) {
    // показывает пользователю заголовок и сообщение в диалоговом окне
};
Controller.prototype.success = function(msg) {
    this.showDialog("Success", msg);
};
Controller.prototype.failure = function(err) {
    this.errors.push(err);
    this.showDialog("Error", err);
};
```

```
// Дочерний класс
function LoginController() {
    Controller.call( this );
}

// Привязываем дочерний класс к родительскому
LoginController.prototype = Object.create( Controller.prototype );
LoginController.prototype.getUser = function() {
    return document.getElementById( "login_username" ).value;
};

LoginController.prototype.getPassword = function() {
    return document.getElementById( "login_password" ).value;
};

LoginController.prototype.validateEntry = function(user,pw) {
    user = user || this.getUser();
    pw = pw || this.getPassword();

    if (!(user && pw)) {
        return this.failure( "Please enter a username & password!" );
    }
    else if (pw.length < 5) {
        return this.failure( "Password must be 5+ characters!" );
    }

    // добрались сюда? валидация прошла успешно!
    return true;
};

// Переопределяем для расширения базового `failure()`
LoginController.prototype.failure = function(err) {
    // вызов "super"
    Controller.prototype.failure.call( this, "Login invalid: " + err );
};

// Дочерний класс
function AuthController(login) {
    Controller.call( this );
    // помимо наследования, нам необходима композиция
    this.login = login;
}

// Привязываем дочерний класс к родительскому
AuthController.prototype = Object.create( Controller.prototype );
AuthController.prototype.server = function(url,data) {
    return $.ajax( {
        url: url,
        data: data
    } );
};

AuthController.prototype.checkAuth = function() {
    var user = this.login.getUser();
    var pw = this.login.getPassword();

    if (this.login.validateEntry( user, pw )) {
        this.server( "/check-auth", {
            ...
        });
    }
};
```

```

        user: user,
        pw: pw
    } )
    .then( this.success.bind( this ) )
    .fail( this.failure.bind( this ) );
}

};

// Переопределяем для расширения базового `success()`
AuthController.prototype.success = function() {
    // вызов "super"
    Controller.prototype.success.call( this, "Authenticated!" );
};

// Переопределяем для расширения базового `failure()`
AuthController.prototype.failure = function(err) {
    // вызов "super"
    Controller.prototype.failure.call( this, "Auth Failed: " + err );
};

var auth = new AuthController(
    // помимо наследования, нам необходима композиция
    new LoginController()
);
auth.checkAuth();

```

У всех контроллеров есть общее базовое поведение: `success(..)`, `failure(..)` и `showDialog(..)`. Дочерние классы `LoginController` и `AuthController` переопределяют `failure(..)` и `success(..)`, дополняя стандартное поведение базового класса. Обратите внимание, что `AuthController` необходим экземпляр `LoginController` для взаимодействия с формой входа, поэтому он становится свойством данных.

Мы также добавили к наследованию немного композиции. `AuthController` должен знать о `LoginController`, поэтому мы создаем экземпляр (`new LoginController()`) и сохраняем ссылку на него в члене данных класса `this.login`, так что `AuthController` может вызывать поведение `LoginController`.

Примечание: Мы могли бы поддаться легкому искушению и унаследовать `AuthController` от `LoginController`, или наоборот, получив виртуальную композицию в цепочке наследования. Но это яркий пример того, какие проблемы порождает наследование классов в качестве модели предметной области. Ведь ни `AuthController`, ни `LoginController` не уточняют поведение друг друга, поэтому наследование между ними не имеет смысла, если только классы не являются вашим единственным шаблоном проектирования. Вместо этого мы добавили простую композицию, и теперь оба класса могут взаимодействовать, сохранив при этом преимущества наследования от базового класса `Controller`.

Если вы разбираетесь в класс-ориентированном (ОО) проектировании, то все это должно выглядеть знакомым и естественным.

Де-класс-ификация

Но действительно ли нам нужно моделировать эту проблему с помощью родительского класса Controller, двух дочерних классов и композиции? Можно ли воспользоваться преимуществами делегирования поведения в стиле OOO и получить гораздо более простой дизайн? Да!

```

var LoginController = {
    errors: [],
    getUser: function() {
        return document.getElementById( "login_username" ).value;
    },
    getPassword: function() {
        return document.getElementById( "login_password" ).value;
    },
    validateEntry: function(user,pw) {
        user = user || this.getUser();
        pw = pw || this.getPassword();

        if (!(user && pw)) {
            return this.failure( "Please enter a username & password!" );
        }
        else if (pw.length < 5) {
            return this.failure( "Password must be 5+ characters!" );
        }

        // добрались сюда? валидация прошла успешно!
        return true;
    },
    showDialog: function(title,msg) {
        // показывает пользователю сообщение об успехе в диалоговом окне
    },
    failure: function(err) {
        this.errors.push( err );
        this.showDialog( "Error", "Login invalid: " + err );
    }
};

// Связываем `AuthController` для делегирования к `LoginController`
var AuthController = Object.create( LoginController );

AuthController.errors = [];
AuthController.checkAuth = function() {
    var user = this.getUser();
    var pw = this.getPassword();

    if (this.validateEntry( user, pw )) {
        this.server( "/check-auth",{
            user: user,
            pw: pw
        } )
        .then( this.accepted.bind( this ) )
        .fail( this.rejected.bind( this ) );
    }
}

```

```

};

AuthController.server = function(url,data) {
    return $.ajax( {
        url: url,
        data: data
    } );
};

AuthController.accepted = function() {
    this.showDialog( "Success", "Authenticated!" )
};

AuthController.rejected = function(err) {
    this.failure( "Auth Failed: " + err );
};

```

Поскольку AuthController теперь просто объект (как и LoginController), нам не нужно создавать экземпляр (`new AuthController()`) для решения нашей задачи. Все, что надо сделать:

```
AuthController.checkAuth();
```

При работе с OOO вы легко можете добавить один или несколько объектов в цепочку делегирования, и вам не придется создавать экземпляры классов:

```

var controller1 = Object.create( AuthController );
var controller2 = Object.create( AuthController );

```

При делегировании поведения AuthController и LoginController являются **просто объектами**, которые находятся на одном уровне и не выстроены в иерархию, как родительские и дочерние классы в класс-ориентированном подходе. Мы выбрали направление делегирования от AuthController к LoginController произвольно — оно вполне могло быть и противоположным.

Основной результат в том, что во втором листинге кода у нас осталось только две сущности (LoginController и AuthController), а **не три** как раньше.

Нам не нужен базовый класс Controller с "общим" поведением для двух других, поскольку делегирование поведения — достаточно мощный механизм, предоставляющий нам все требуемую функциональность. Нам также не нужно создавать экземпляры классов, поскольку классов нет, а есть **лишь сами объекты**. Более того, нет нужды в композиции, потому что делегирование позволяет обоим объектам взаимодействовать как угодно.

Наконец, мы избежали ловушек с полиморфизмом в класс-ориентированном дизайне, отказавшись от одинаковых имен `success(..)` и `failure(..)` в обоих классах, ведь иначе нам потребовался бы уродливый явный псевдополиморфизм. Вместо этого, мы назвали их `accepted()` и `rejected(..)` в AuthController, и эти имена немного лучше описывают выполняемые задачи.

Подведем итог: мы получили ту же функциональность, но (гораздо) более простой дизайн. В этом и состоит мощь OOO-стиля и шаблона проектирования *делегирования поведения*.

› Более элегантный синтаксис

Одно из приятных новшеств, которое делает `class` в ES6 обманчиво притягательным (о том, почему стоит его избегать, см. в Приложении A!), — сокращенный синтаксис для объявления методов класса:

```
class Foo {
    methodName() { /* .. */ }
}
```

Мы избавились от ключевого слова `function` в объявлении, что обрадовало JS-разработчиков по всему миру!

Вы наверное заметили, что в OOO синтаксисе `function` встречается на каждом шагу, что немного расходится с нашей целью упростить код. **Но мы можем это исправить!**

В ES6 мы можем использовать *сокращенные объявления методов* в любом объектном литерале, поэтому объект в OOO-стиле можно объявить так (такой же сокращенный синтаксис, что и в теле `class`):

```
var LoginController = {
    errors: [],
    getUser() { // Смотри-ка, нет `function`!
        // ...
    },
    getPassword() {
        // ...
    }
    // ...
};
```

Единственная разница в том, что в объектных литералах по-прежнему надо использовать разделители `,` между элементами, тогда как синтаксис `class` этого не требует. Но на фоне общей картины это сущий пустяк.

Более того, в ES6 мы можем заменить неуклюжий синтаксис с отдельным присваиванием каждого свойства (как в определении `AuthController`) на объектный литерал (с помощью сокращенной формы записи методов), и изменить `[[Prototype]]` этого объекта на `Object.setPrototypeOf(..)`:

```
// используем более красивый синтаксис объектного литерала
// с краткими методами!
var AuthController = {
    errors: [],
    checkAuth() {
        // ...
    },
    server(url,data) {
        // ...
    }
};
```

```

    }
    // ...
};

// ТЕПЕРЬ, свяжем `AuthController` через делегирование с `LoginController`
Object.setPrototypeOf( AuthController, LoginController );

```

С краткими методами ES6 наш OOOO-стиль **стал еще более удобным** чем раньше (но даже без этого он и так был гораздо проще и симпатичнее чем классический прототипный код). Чтобы получить красивый и чистый объектный синтаксис, вам **не нужны классы!**

› Лексический недостаток

У кратких методов есть небольшой недостаток, о котором нужно знать. Рассмотрим код:

```

var Foo = {
  bar() { /*..*/ },
  baz: function baz() { /*..*/ }
};

```

Если убрать синтаксический сахар, то этот код будет работать так:

```

var Foo = {
  bar: function() { /*..*/ },
  baz: function baz() { /*..*/ }
};

```

Видите разницу? Сокращенный вариант `bar()` превратился в *анонимное функциональное выражение* (`function()..`), привязанное к свойству `bar`, поскольку у объекта функции нет имени. Сравните это с указанным вручную *именованным функциональным выражением* (`function baz()..`), которое не только привязано к свойству `.baz`, но и имеет лексический идентификатор `baz`.

И что из этого? В книге "Область видимости и замыкания" нашей серии "Вы не знаете JS", мы подробно рассмотрели три основных недостатка *анонимных функциональных выражений*. Перечислим их еще раз и посмотрим, что из этого затрагивает краткие методы.

Отсутствие идентификатора `name` у анонимной функции:

1. усложняет отладку стектрейсов (stack traces)
2. усложняет работу с функциями, ссылающимися на самих себя (рекурсия, подписка/отписка обработчика события, и т.п.)
3. немного затрудняет понимание кода

Пункты 1 и 3 не относятся к кратким методам.

Несмотря на то, что код без синтаксического сахара превращается в *анонимное функциональное выражение*, у которого обычно нет `name` в стектрейсах, краткие методы

имеют внутреннее свойство `name` для объекта функции, поэтому стектрейсы могут его использовать (хотя это зависит от реализации и не гарантируется).

Пункт 2, к сожалению, является недостатком кратких методов. У них нет лексического идентификатора, на который они могли бы ссылаться. Рассмотрим:

```
var Foo = {
    bar: function(x) {
        if (x < 10) {
            return Foo.bar( x * 2 );
        }
        return x;
    },
    baz: function baz(x) {
        if (x < 10) {
            return baz( x * 2 );
        }
        return x;
    }
};
```

Явная ссылка `Foo.bar(x*2)` в этом примере вроде бы решает проблему, но во многих случаях у функции нет такой возможности, например когда функция совместно используется различными объектами через делегирование, когда используется привязка `this`, и т.п. Вам придется использовать реальную ссылку функции на саму себя, и лучший способ сделать это — идентификатор `name` объекта функции.

Просто помните об этой особенности кратких методов, и если возникнет проблема со ссылкой функции на саму себя, откажитесь от краткого синтаксиса **в данном конкретном объявлении** метода в пользу *именованного функционального выражения*: `baz: function baz() { .. }.`

Интроспекция

Если вы долгое время писали программы в класс-ориентированном стиле (в JS или других языках), то наверняка знаете, что такое *интроспекция типа*: проверка экземпляра с целью выяснить, какого вида объект перед вами. Основная цель *интроспекции типа* экземпляра класса — узнать о структуре и функциональных возможностях объекта исходя из того как он был создан.

Рассмотрим пример кода, в котором для интроспекции объекта `a1` используется `instanceof` (см. главу 5):

```
function Foo() {
    // ...
}
Foo.prototype.something = function(){
    // ...
}
```

```
var a1 = new Foo();

// позднее

if (a1 instanceof Foo) {
    a1.something();
}
```

Благодаря `Foo.prototype` (не `Foo!`) в цепочке `[[Prototype]]` (см. главу 5) объекта `a1`, оператор `instanceof` сообщает нам, что `a1` будто бы является экземпляром "класса" `Foo`. Исходя из этого мы предполагаем, что у `a1` есть функциональные возможности, описанные в "классе" `Foo`.

Разумеется, никакого класса `Foo` не существует, есть всего-навсего обычная функция `Foo`, у которой есть ссылка на некоторый объект (`Foo.prototype`), с которым `a1` связывается ссылкой делегирования. По идее, оператор `instanceof` исходя из его названия должен проверять взаимосвязь между `a1` и `Foo`, но на самом деле он лишь сообщает нам, связаны ли `a1` и некий объект, на который ссылается `Foo.prototype`.

Семантическая путаница (и косвенность) синтаксиса `instanceof` приводит к тому, что для интроспекции объекта `a1` с целью выяснить, обладает ли он функциональными возможностями искомого объекта, вам *необходима* функция, содержащая ссылку на этот объект. То есть, вы не можете напрямую узнать, связаны ли два объекта.

Вспомните абстрактный пример `Foo / Bar / b1`, который мы рассматривали ранее в этой главе:

```
function Foo() { /* .. */ }
Foo.prototype...

function Bar() { /* .. */ }
Bar.prototype = Object.create( Foo.prototype );

var b1 = new Bar( "b1" );
```

Вот список проверок, которые вам придется выполнить для *интроспекции типов* этих сущностей с помощью семантики `instanceof` и `.prototype`:

```
// устанавливаем связь между `Foo` и `Bar`
Bar.prototype instanceof Foo; // true
Object.getPrototypeOf( Bar.prototype ) === Foo.prototype; // true
Foo.prototype.isPrototypeOf( Bar.prototype ); // true

// устанавливаем связь между `b1` и `Foo` и `Bar`
b1 instanceof Foo; // true
b1 instanceof Bar; // true
Object.getPrototypeOf( b1 ) === Bar.prototype; // true
Foo.prototype.isPrototypeOf( b1 ); // true
Bar.prototype.isPrototypeOf( b1 ); // true
```

Согласитесь, что это немного отстойно. Например, интуитивно хочется, чтобы была возможность написать что-то вроде `Bar instanceof Foo` (потому что "instance" — довольно широкое понятие, и можно подумать, что оно включает в себя и "наследование"). Но такое сравнение в JS бессмысленно. Вместо этого приходится использовать `Bar.prototype instanceof Foo`.

Еще один распространенный, но возможно менее надежный метод *интроспекции типов*, который многие разработчики предпочитают оператору `instanceof`, называется "утиная типизация". Этот термин берет свое начало из афоризма "если нечто выглядит как утка и крякает как утка, то возможно это и есть утка".

Пример:

```
if (a1.something) {
    a1.something();
}
```

Вместо того, чтобы проверять связь между `a1` и объектом, содержащим делегируемую функцию `something()`, мы предполагаем, что успешная проверка `a1.something` означает, что `a1` позволяет вызывать `.something()` (неважно, найден ли метод непосредственно в `a1` или делегирован какому-то другому объекту). Само по себе это предположение не такое уж и рискованное.

Однако зачастую понятие "утиной типизации" расширяется, и делаются **дополнительные предположения о возможностях объекта**, выходящие за рамки проверки. Разумеется, это увеличивает риски и делает дизайн более хрупким.

Ярким примером "утиной типизации" являются обещания (Promises) в ES6 (как мы уже говорили, их рассмотрение выходит за рамки этой книги).

В ряде случаев возникает необходимость проверить, является ли ссылка на некий объект обещанием, причем это делается путем проверки наличия у объекта функции `then()`. Другими словами, если у **какого угодно** объекта найдется метод `then()`, то механизм обещаний ES6 будет считать что это "**thenable**" объект, и будет ожидать от него стандартного поведения Promises.

Если у вас какой-либо не-Promise объект, у которого по какой-то причине есть метод `then()`, то настоятельно рекомендуется держать его подальше от механизма ES6 Promise, чтобы избежать некорректных предположений.

Этот пример наглядно иллюстрирует риски "утиной типизации". Подобные вещи следует использовать лишь в разумных пределах и в контролируемом окружении.

Возвращаясь к коду в стиле OOO отметим, что *интроспекция типов* в данном случае может быть гораздо элегантнее. Давайте вспомним фрагмент OOO кода `Foo / Bar / b1`, рассмотренный ранее в этой главе:

```
var Foo = { /* .. */ };

var Bar = Object.create( Foo );
```

Bar...

```
var b1 = Object.create( Bar );
```

Поскольку в OOO у нас есть лишь обычные объекты, связанные делегированием [[Prototype]], мы можем использовать гораздо более простую форму *интроспекции типов*:

```
// устанавливаем связь между `Foo` и `Bar`  
Foo.isPrototypeOf( Bar ); // true  
Object.getPrototypeOf( Bar ) === Foo; // true  
  
// устанавливаем связь между `b1` и `Foo` и `Bar`  
Foo.isPrototypeOf( b1 ); // true  
Bar.isPrototypeOf( b1 ); // true  
Object.getPrototypeOf( b1 ) === Bar; // true
```

Мы больше не используем instanceof, потому что он претендует на то, что каким-то образом связан с классами. Теперь мы просто задаем (неформальный) вопрос "являешься ли ты моим прототипом?" Больше не нужны косвенные обращения, такие как Foo.prototype или ужасно многословное Foo.prototype.isPrototypeOf(...).

Я считаю, что эти проверки гораздо более простые и не такие запутанные, как предыдущий набор интроспектирующих проверок. И снова мы видим, что в JavaScript подход OOO проще, чем кодирование в стиле классов (и при этом обладает теми же возможностями).

’Обзор (TL;DR)

Классы и наследование — это один из возможных шаблонов проектирования, который вы можете использовать или не использовать в архитектуре вашего ПО. Большинство разработчиков считают само собой разумеющимся тот факт, что классы являются единственным (правильным) способом организации кода. Но в этой главе мы увидели другой, менее популярный, но весьма мощный шаблон проектирования: **делегирование поведения**.

Делегирование поведения предполагает, что все объекты находятся на одном уровне и связаны друг с другом делегированием, а не отношениями родитель-потомок.

Механизм [[Prototype]] в JavaScript по своему замыслу является механизмом делегирования поведения. Это значит, что мы можем либо всячески пытаться реализовать механику классов поверх JS (см. главы 4 и 5), либо принять истинную сущность [[Prototype]] как механизма делегирования.

Если вы проектируете код, используя только объекты, это не только упрощает синтаксис, но и позволяет добиться более простой архитектуры кода.

OOO (объекты, связанные с другими объектами) — это стиль кодирования, в котором объекты создаются и связываются друг с другом без абстракции классов. OOO вполне естественным образом реализует делегирование поведения при помощи [[Prototype]].

Appendix A: ES6 class

If there's any take-away message from the second half of this book (Chapters 4-6), it's that classes are an optional design pattern for code (not a necessary given), and that furthermore they are often quite awkward to implement in a [[Prototype]] language like JavaScript.

This awkwardness is *not* just about syntax, although that's a big part of it. Chapters 4 and 5 examined quite a bit of syntactic ugliness, from verbosity of `.prototype` references cluttering the code, to *explicit pseudo-polymorphism* (see Chapter 4) when you give methods the same name at different levels of the chain and try to implement a polymorphic reference from a lower-level method to a higher-level method. `.constructor` being wrongly interpreted as "was constructed by" and yet being unreliable for that definition is yet another syntactic ugly.

But the problems with class design are much deeper. Chapter 4 points out that classes in traditional class-oriented languages actually produce a *copy* action from parent to child to instance, whereas in [[Prototype]], the action is **not** a copy, but rather the opposite -- a delegation link.

When compared to the simplicity of OOO-style code and behavior delegation (see Chapter 6), which embrace [[Prototype]] rather than hide from it, classes stand out as a sore thumb in JS.

’class

But we *don't* need to re-argue that case again. I re-mention those issues briefly only so that you keep them fresh in your mind now that we turn our attention to the ES6 `class` mechanism. We'll demonstrate here how it works, and look at whether or not `class` does anything substantial to address any of those "class" concerns.

Let's revisit the `Widget` / `Button` example from Chapter 6:

```
class Widget {
    constructor(width,height) {
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    }
    render($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            }).appendTo( $where );
        }
    }
}

class Button extends Widget {
    constructor(width,height,label) {
        super( width, height );
    }
}
```

```

        this.label = label || "Default";
        this.$elem = $( "<button>" ).text( this.label );
    }
    render($where) {
        super.render( $where );
        this.$elem.click( this.onClick.bind( this ) );
    }
    onClick(evt) {
        console.log( "Button '" + this.label + "' clicked!" );
    }
}

```

Beyond this syntax *looking* nicer, what problems does ES6 solve?

1. There's no more (well, sorta, see below!) references to `.prototype` cluttering the code.
2. `Button` is declared directly to "inherit from" (aka extends) `Widget`, instead of needing to use `Object.create(..)` to replace a `.prototype` object that's linked, or having to set with `.__proto__` or `Object.setPrototypeOf(..)`.
3. `super(..)` now gives us a very helpful **relative polymorphism** capability, so that any method at one level of the chain can refer relatively one level up the chain to a method of the same name. This includes a solution to the note from Chapter 4 about the weirdness of constructors not belonging to their class, and so being unrelated -- `super()` works inside constructors exactly as you'd expect.
4. `class` literal syntax has no affordance for specifying properties (only methods). This might seem limiting to some, but it's expected that the vast majority of cases where a property (state) exists elsewhere but the end-chain "instances", this is usually a mistake and surprising (as it's state that's implicitly "shared" among all "instances"). So, one *could* say the `class` syntax is protecting you from mistakes.
5. `extends` lets you extend even built-in object (sub)types, like `Array` or `RegExp`, in a very natural way. Doing so without `class .. extends` has long been an exceedingly complex and frustrating task, one that only the most adept of framework authors have ever been able to accurately tackle. Now, it will be rather trivial!

In all fairness, those are some substantial solutions to many of the most obvious (syntactic) issues and surprises people have with classical prototype-style code.

’ `class` Gotchas

It's not all bubblegum and roses, though. There are still some deep and profoundly troubling issues with using "classes" as a design pattern in JS.

Firstly, the `class` syntax may convince you a new "class" mechanism exists in JS as of ES6. **Not so.** `class` is, mostly, just syntactic sugar on top of the existing `[[Prototype]]` (delegation!) mechanism.

That means `class` is not actually copying definitions statically at declaration time the way it does in traditional class-oriented languages. If you change/replace a method (on purpose or by accident) on the parent "class", the child "class" and/or instances will still be "affected", in that they didn't get copies at declaration time, they are all still using the live-delegation model based on `[[Prototype]]`:

```

class C {
    constructor() {
        this.num = Math.random();
    }
    rand() {
        console.log( "Random: " + this.num );
    }
}

var c1 = new C();
c1.rand(); // "Random: 0.4324299..."

C.prototype.rand = function() {
    console.log( "Random: " + Math.round( this.num * 1000 ) );
};

var c2 = new C();
c2.rand(); // "Random: 867"

c1.rand(); // "Random: 432" -- oops!!!

```

This only seems like reasonable behavior *if you already know* about the delegation nature of things, rather than expecting *copies* from "real classes". So the question to ask yourself is, why are you choosing `class` syntax for something fundamentally different from classes?

Doesn't the ES6 `class` syntax **just make it harder** to see and understand the difference between traditional classes and delegated objects?

`class` syntax *does not* provide a way to declare class member properties (only methods). So if you need to do that to track shared state among instances, then you end up going back to the ugly `.prototype` syntax, like this:

```

class C {
    constructor() {
        // make sure to modify the shared state,
        // not set a shadowed property on the
        // instances!
        C.prototype.count++;

        // here, `this.count` works as expected
        // via delegation
        console.log( "Hello: " + this.count );
    }
}

// add a property for shared state directly to
// prototype object
C.prototype.count = 0;

var c1 = new C();
// Hello: 1

```

```
var c2 = new C();
// Hello: 2

c1.count === 2; // true
c1.count === c2.count; // true
```

The biggest problem here is that it betrays the `class` syntax by exposing (leakage!) `.prototype` as an implementation detail.

But, we also still have the surprise gotcha that `this.count++` would implicitly create a separate shadowed `.count` property on both `c1` and `c2` objects, rather than updating the shared state. `class` offers us no consolation from that issue, except (presumably) to imply by lack of syntactic support that you shouldn't be doing that *at all*.

Moreover, accidental shadowing is still a hazard:

```
class C {
    constructor(id) {
        // oops, gotcha, we're shadowing `id()` method
        // with a property value on the instance
        this.id = id;
    }
    id() {
        console.log("Id: " + this.id);
    }
}

var c1 = new C("c1");
c1.id(); // TypeError -- `c1.id` is now the string "c1"
```

There's also some very subtle nuanced issues with how `super` works. You might assume that `super` would be bound in an analogous way to how `this` gets bound (see Chapter 2), which is that `super` would always be bound to one level higher than whatever the current method's position in the `[[Prototype]]` chain is.

However, for performance reasons (`this` binding is already expensive), `super` is not bound dynamically. It's bound sort of "statically", as declaration time. No big deal, right?

Ehh... maybe, maybe not. If you, like most JS devs, start assigning functions around to different objects (which came from `class` definitions), in various different ways, you probably won't be very aware that in all those cases, the `super` mechanism under the covers is having to be re-bound each time.

And depending on what sorts of syntactic approaches you take to these assignments, there may very well be cases where the `super` can't be properly bound (at least, not where you suspect), so you may (at time of writing, TC39 discussion is ongoing on the topic) have to manually bind `super` with `toMethod(..)` (kinda like you have to do `bind(..)` for `this` -- see Chapter 2).

You're used to being able to assign around methods to different objects to *automatically* take advantage of the dynamism of `this` via the *implicit binding* rule (see Chapter 2). But the same will likely not be true with methods that use `super`.

Consider what `super` should do here (against D and E):

```

class P {
    foo() { console.log( "P.foo" ); }
}

class C extends P {
    foo() {
        super();
    }
}

var c1 = new C();
c1.foo(); // "P.foo"

var D = {
    foo: function() { console.log( "D.foo" ); }
};

var E = {
    foo: C.prototype.foo
};

// Link E to D for delegation
Object.setPrototypeOf( E, D );

E.foo(); // "P.foo"

```

If you were thinking (quite reasonably!) that `super` would be bound dynamically at call-time, you might expect that `super()` would automatically recognize that E delegates to D, so `E.foo()` using `super()` should call to `D.foo()`.

Not so. For performance pragmatism reasons, `super` is not *late bound* (aka, dynamically bound) like this is. Instead it's derived at call-time from `[[HomeObject]].[[Prototype]]`, where `[[HomeObject]]` is statically bound at creation time.

In this particular case, `super()` is still resolving to `P.foo()`, since the method's `[[HomeObject]]` is still `c` and `C.[[Prototype]]` is `P`.

There will *probably* be ways to manually address such gotchas. Using `toMethod(..)` to bind/rebind a method's `[[HomeObject]]` (along with setting the `[[Prototype]]` of that object!) appears to work in this scenario:

```

var D = {
    foo: function() { console.log( "D.foo" ); }
};

// Link E to D for delegation
var E = Object.create( D );

// manually bind `foo`'s `[[HomeObject]]` as
// `E`, and `E.[[Prototype]]` is `D`, so thus

```

```
// `super()` is `D.foo()`  
E.foo = C.prototype.foo.toMethod( E, "foo" );  
  
E.foo(); // "D.foo"
```

Note: `toMethod(..)` clones the method, and takes `homeObject` as its first parameter (which is why we pass `E`), and the second parameter (optionally) sets a name for the new method (which keep at "foo").

It remains to be seen if there are other corner case gotchas that devs will run into beyond this scenario. Regardless, you will have to be diligent and stay aware of which places the engine automatically figures out `super` for you, and which places you have to manually take care of it. Ugh!

‣ Static > Dynamic?

But the biggest problem of all about ES6 `class` is that all these various gotchas mean `class` kinda opts you into a syntax which seems to imply (like traditional classes) that once you declare a `class`, it's a static definition of a (future instantiated) thing. You completely lose sight of the fact `c` is an object, a concrete thing, which you can directly interact with.

In traditional class-oriented languages, you never adjust the definition of a class later, so the class design pattern doesn't suggest such capabilities. But **one of the most powerful parts** of JS is that it *is* dynamic, and the definition of any object is (unless you make it immutable) a fluid and mutable *thing*.

`class` seems to imply you shouldn't do such things, by forcing you into the uglier `.prototype` syntax to do so, or forcing you think about `super` gotchas, etc. It also offers *very little* support for any of the pitfalls that this dynamism can bring.

In other words, it's as if `class` is telling you: "dynamic is too hard, so it's probably not a good idea. Here's a static-looking syntax, so code your stuff statically."

What a sad commentary on JavaScript: **dynamic is too hard, let's pretend to be (but not actually be!) static.**

These are the reasons why ES6 `class` is masquerading as a nice solution to syntactic headaches, but it's actually muddying the waters further and making things worse for JS and for clear and concise understanding.

Note: If you use the `.bind(..)` utility to make a hard-bound function (see Chapter 2), the function created is not subclassable with ES6 `extend` like normal functions are.

‣ Review (TL;DR)

`class` does a very good job of pretending to fix the problems with the class/inheritance design pattern in JS. But it actually does the opposite: **it hides many of the problems, and introduces other subtle but dangerous ones.**

`class` contributes to the ongoing confusion of "class" in JavaScript which has plagued the language for nearly two decades. In some respects, it asks more questions than it answers, and it feels in totality like a very unnatural fit on top of the elegant simplicity of the `[[Prototype]]` mechanism.

Bottom line: if ES6 `class` makes it harder to robustly leverage `[[Prototype]]`, and hides the most important nature of the JS object mechanism -- **the live delegation links between objects** -- shouldn't we see `class` as creating more troubles than it solves, and just relegate it to an anti-pattern?

I can't really answer that question for you. But I hope this book has fully explored the issue at a deeper level than you've ever gone before, and has given you the information you need *to answer it yourself*.

Appendix B: Acknowledgments

I have many people to thank for making this book title and the overall series happen.

First, I must thank my wife Christen Simpson, and my two kids Ethan and Emily, for putting up with Dad always pecking away at the computer. Even when not writing books, my obsession with JavaScript glues my eyes to the screen far more than it should. That time I borrow from my family is the reason these books can so deeply and completely explain JavaScript to you, the reader. I owe my family everything.

I'd like to thank my editors at O'Reilly, namely Simon St.Laurent and Brian MacDonald, as well as the rest of the editorial and marketing staff. They are fantastic to work with, and have been especially accommodating during this experiment into "open source" book writing, editing, and production.

Thank you to the many folks who have participated in making this book series better by providing editorial suggestions and corrections, including Shelley Powers, Tim Ferro, Evan Borden, Forrest L. Norvell, Jennifer Davis, Jesse Harlin, and many others. A big thank you to Nick Berardi for writing the Foreword for this title.

Thank you to the countless folks in the community, including members of the TC39 committee, who have shared so much knowledge with the rest of us, and especially tolerated my incessant questions and explorations with patience and detail. John-David Dalton, Jurij "kangax" Zaytsev, Mathias Bynens, Axel Rauschmayer, Nicholas Zakas, Angus Croll, Reginald Braithwaite, Dave Herman, Brendan Eich, Allen Wirfs-Brock, Bradley Meck, Domenic Denicola, David Walsh, Tim Disney, Peter van der Zee, Andrea Giammarchi, Kit Cambridge, Eric Elliott, and so many others, I can't even scratch the surface.

The *You Don't Know JS* book series was born on Kickstarter, so I also wish to thank all my (nearly) 500 generous backers, without whom this book series could not have happened:

Jan Szpila, nokiko, Murali Krishnamoorthy, Ryan Joy, Craig Patchett, pdqtrader, Dale Fukami, ray hatfield, R0drigo Perez [Mx], Dan Petitt, Jack Franklin, Andrew Berry, Brian Grinstead, Rob Sutherland, Sergi Meseguer, Phillip Gourley, Mark Watson, Jeff Carouth, Alfredo Sumaran, Martin Sachse, Marcio Barrios, Dan, AimelyneM, Matt Sullivan, Delnatte Pierre-Antoine, Jake Smith, Eugen Tudorancea, Iris, David Trinh, simonstl, Ray Daly, Uros Gruber, Justin Myers, Shai Zonis, Mom & Dad, Devin Clark, Dennis Palmer, Brian Panahi Johnson, Josh Marshall, Marshall, Dennis Kerr, Matt Steele, Erik Slagter, Sacah, Justin Rainbow, Christian Nilsson, Delapouite, D.Pereira, Nicolas Hoizey, George V. Reilly, Dan Reeves, Bruno Laturner, Chad Jennings, Shane King, Jeremiah Lee Cohick, od3n, Stan Yamane, Marko Vucinic, Jim B, Stephen Collins, Ægir Porsteinsson, Eric Pederson, Owain, Nathan Smith, Jeanetteurphy, Alexandre ELISÉ, Chris Peterson, Rik Watson, Luke Matthews, Justin Lowery, Morten Nielsen, Vernon Kesner, Chetan Shenoy, Paul Treguing, Marc Grabanski, Dion Almaer, Andrew Sullivan, Keith Elsass, Tom Burke, Brian Ashenfelter, David Stuart, Karl Swedberg, Graeme, Brandon Hays, John Christopher, Gior, manoj reddy, Chad Smith, Jared Harbour, Minoru TODA, Chris Wigley, Daniel Mee, Mike, Handyface, Alex Jahraus, Carl Furrow, Rob Foulkrod, Max Shishkin, Leigh Penny Jr., Robert Ferguson, Mike van Hoenselaar, Hasse Schougaard, rajan venkataguru, Jeff Adams, Trae Robbins, Rolf Langenhuijzen, Jorge Antunes, Alex Koloskov, Hugh Greenish, Tim Jones, Jose

Ochoa, Michael Brennan-White, Naga Harish Muvva, Barkóczi Dávid, Kitt Hodsdon, Paul McGraw, Sascha Goldhofer, Andrew Metcalf, Markus Krogh, Michael Mathews, Matt Jared, Juanfran, Georgie Kirschner, Kenny Lee, Ted Zhang, Amit Pahwa, Inbal Sinai, Dan Raine, Schabse Laks, Michael Tervoort, Alexandre Abreu, Alan Joseph Williams, NicolasD, Cindy Wong, Reg Braithwaite, LocalPCGuy, Jon Friskics, Chris Merriman, John Pena, Jacob Katz, Sue Lockwood, Magnus Johansson, Jeremy Crapsey, Grzegorz Pawłowski, nico nuzzaci, Christine Wilks, Hans Bergren, charles montgomery, Ariel בבל-רְגָב Fogel, Ivan Kolev, Daniel Campos, Hugh Wood, Christian Bradford, Frédéric Harper, Ionuț Dan Popa, Jeff Trimble, Rupert Wood, Trey Carrico, Pancho Lopez, Joël kuijten, Tom A Marra, Jeff Jewiss, Jacob Rios, Paolo Di Stefano, Soledad Penades, Chris Gerber, Andrey Dolganov, Wil Moore III, Thomas Martineau, Kareem, Ben Thouret, Udi Nir, Morgan Laupies, jory carson-burson, Nathan L Smith, Eric Damon Walters, Derry Lozano-Hoyland, Geoffrey Wiseman, mkeehner, KatieK, Scott MacFarlane, Brian LaShomb, Adrien Mas, christopher ross, Ian Littman, Dan Atkinson, Elliot Jobe, Nick Dozier, Peter Wooley, John Hoover, dan, Martin A. Jackson, Héctor Fernando Hurtado, andy ennamorato, Paul Seltmann, Melissa Gore, Dave Pollard, Jack Smith, Philip Da Silva, Guy Israeli, @megalithic, Damian Crawford, Felix Gliesche, April Carter Grant, Heidi, jim tierney, Andrea Giammarchi, Nico Vignola, Don Jones, Chris Hartjes, Alex Howes, john gibbon, David J. Groom, BBox, Yu 'Dilys' Sun, Nate Steiner, Brandon Satrom, Brian Wyant, Wesley Hales, Ian Pouncey, Timothy Kevin Oxley, George Terezakis, sanjay raj, Jordan Harband, Marko McLion, Wolfgang Kaufmann, Pascal Peuckert, Dave Nugent, Markus Liebelt, Welling Guzman, Nick Cooley, Daniel Mesquita, Robert Syvarth, Chris Coyier, Rémy Bach, Adam Dougal, Alistair Duggin, David Loidolt, Ed Richer, Brian Chenault, GoldFire Studios, Carles Andrés, Carlos Cabo, Yuya Saito, roberto ricardo, Barnett Klane, Mike Moore, Kevin Marx, Justin Love, Joe Taylor, Paul Dijou, Michael Kohler, Rob Cassie, Mike Tierney, Cody Leroy Lindley, tofuji, Shimon Schwartz, Raymond, Luc De Brouwer, David Hayes, Rhys Brett-Bowen, Dmitry, Aziz Khoury, Dean, Scott Tolinski - Level Up, Clement Boirie, Djordje Lukic, Anton Kotenko, Rafael Corral, Philip Hurwitz, Jonathan Pidgeon, Jason Campbell, Joseph C., SwiftOne, Jan Hohner, Derick Bailey, getify, Daniel Cousineau, Chris Charlton, Eric Turner, David Turner, Joël Galeran, Dharma Vagabond, adam, Dirk van Bergen, dave ❤️🎶★ furf, Vedran Zakanj, Ryan McAllen, Natalie Patrice Tucker, Eric J. Bivona, Adam Spooner, Aaron Cavano, Kelly Packer, Eric J, Martin Drenovac, Emilis, Michael Pelikan, Scott F. Walter, Josh Freeman, Brandon Hudgeons, vijay chennupati, Bill Glennon, Robin R., Troy Forster, otaku_coder, Brad, Scott, Frederick Ostrander, Adam Brill, Seb Flippence, Michael Anderson, Jacob, Adam Randlett, Standard, Joshua Clanton, Sebastian Kouba, Chris Deck, SwordFire, Hannes Papenberg, Richard Woeber, hnzz, Rob Crowther, Jedidiah Broadbent, Sergey Chernyshev, Jay-Ar Jamon, Ben Combee, luciano bonachela, Mark Tomlinson, Kit Cambridge, Michael Melgares, Jacob Adams, Adrian Bruinhout, Bev Wieber, Scott Puleo, Thomas Herzog, April Leone, Daniel Mizieliński, Kees van Ginkel, Jon Abrams, Erwin Heiser, Avi Laviad, David newell, Jean-Francois Turcot, Niko Roberts, Erik Dana, Charles Neill, Aaron Holmes, Grzegorz Ziolkowski, Nathan Youngman, Timothy, Jacob Mather, Michael Allan, Mohit Seth, Ryan Ewing, Benjamin Van Treese, Marcelo Santos, Denis Wolf, Phil Keys, Chris Yung, Timo Tijhof, Martin Lekvall, Agendine, Greg Whitworth, Helen Humphrey, Dougal Campbell, Johannes Harth, Bruno Girin, Brian Hough, Darren Newton, Craig McPheat, Olivier Tille, Dennis Roethig, Mathias Bynens, Brendan Stromberger, sundeep, John Meyer, Ron Male, John F Croston III, gigante, Carl Bergenhem, B.J. May, Rebekah Tyler, Ted Foxberry, Jordan Reese, Terry Suitor, afeliz, Tom Kiefer, Darragh Duffy, Kevin Vanderbeken, Andy Pearson, Simon Mac Donald, Abid Din, Chris Joel, Tomas Theunissen, David Dick, Paul Grock, Brandon Wood, John Weis, dgrebb, Nick Jenkins, Chuck Lane, Johnny Megahan, marzsman, Tatu Tamminen, Geoffrey Knauth, Alexander Tarmolov, Jeremy Tymes, Chad Auld, Sean Parmelee, Rob Staenke, Dan Bender, Yannick derwa, Joshua Jones, Geert Plaisier, Tom LeZotte, Christen Simpson,

Stefan Bruvik, Justin Falcone, Carlos Santana, Michael Weiss, Pablo Villoslada, Peter deHaan, Dimitris Iliopoulos, seyDoggy, Adam Jordens, Noah Kantrowitz, Amol M, Matthew Winnard, Dirk Ginader, Phinam Bui, David Rapson, Andrew Baxter, Florian Bougel, Michael George, Alban Escalier, Daniel Sellers, Sasha Rudan, John Green, Robert Kowalski, David I. Teixeira (@ditma, Charles Carpenter, Justin Yost, Sam S, Denis Ciccale, Kevin Sheurs, Yannick Croissant, Pau Fracés, Stephen McGowan, Shawn Searcy, Chris Ruppel, Kevin Lamping, Jessica Campbell, Christopher Schmitt, Sablons, Jonathan Reisdorf, Bunni Gek, Teddy Huff, Michael Mullany, Michael Fürstenberg, Carl Henderson, Rick Yoesting, Scott Nichols, Hernán Ciudad, Andrew Maier, Mike Stapp, Jesse Shawl, Sérgio Lopes, jsulak, Shawn Price, Joel Clermont, Chris Ridmann, Sean Timm, Jason Finch, Aiden Montgomery, Elijah Manor, Derek Gathright, Jesse Harlin, Dillon Curry, Courtney Myers, Diego Cadenas, Arne de Bree, João Paulo Dubas, James Taylor, Philipp Kraeutli, Mihai Păun, Sam Gharegozlu, joshjs, Matt Murchison, Eric Windham, Timo Behrmann, Andrew Hall, joshua price, Théophile Villard

This book series is being produced in an open source fashion, including editing and production. We owe GitHub a debt of gratitude for making that sort of thing possible for the community!

Thank you again to all the countless folks I didn't name but who I nonetheless owe thanks. May this book series be "owned" by all of us and serve to contribute to increasing awareness and understanding of the JavaScript language, to the benefit of all current and future community contributors.