Глава 16

Как работает this

●○○○○

> Я изобрел термин «объектно-ориентированный»
> и могу сказать вам, что при этом не имел в виду C++.
>
> *Алан Кэй (Alan Kay)*

Язык Self — это диалект языка Smalltalk, в котором классы заменены прототипами. Объект мог наследоваться непосредственно из другого объекта. Классическая модель страдала от нестабильности и раздувания по причине сильного сцепления классов посредством расширений `extends`. Прототипы языка Self были весьма удачным упрощением. Модель прототипов легче и выразительнее.

В JavaScript реализована весьма странная модель прототипов.

Когда объект создан, может быть назначен прототип, состоящий из части или всего содержимого нового объекта:

```
const new_object = Object.create(old_object);
```

Объекты, по сути, всего лишь контейнеры для свойств, а прототипы — это просто объекты. Методы — это всего лишь функции, сохраненные в объектах.

При попытке извлечения значения свойства, которого нет у объекта, получается значение `undefined`. Но если у объекта имеется прототип (как у показанного ранее `new_object`), в результате получается значение свойства прототипа. Если и его извлечь не удается и если у прототипа есть свой прототип, получается значение свойства прототипа, принадлежащего первому прототипу. И все спускается ниже и ниже.

У многих объектов может быть общий прототип. Эти объекты могут рассматриваться как экземпляры класса, но, по сути, это всего лишь отдельные объекты с общим прототипом.

Чаще всего прототипы используются в качестве места хранения методов. У схожих объектов, скорее всего, имеются одинаковые методы, поэтому, если методы помещаются в один общий прототип, а не в каждый объект, можно сэкономить память.

А как функция в прототипе узнает, с каким объектом она работает? Здесь на сцену выходит привязка `this`.

Синтаксически вызов метода представляет собой тернарную операцию, использующую `.` и вызов `()` или ссылку `[]` и вызов `()`. Тремя подвыражениями в тернарном выражении являются:

- интересующий объект;
- имя метода;
- список аргументов.

Именованный метод ищут в объекте и в цепочке его прототипов. Если функция не найдена, выдается исключение. Это замечательно и подталкивает к использованию полиморфизма. Наследственность объекта вас волновать не должна. Нужно лишь побеспокоиться о его возможностях. Если возможностей у объекта нет, выдается исключение. Если они есть, мы ими пользуемся, не беспокоясь о том, каким образом они были приобретены.

Если функция найдена, она вызывается со списком аргументов. Функция также получает в качестве подразумеваемого параметр `this`, привязанный к интересующему объекту.

Если метод содержит внутреннюю функцию, последняя не получает доступа к `this`, поскольку внутренние функции вызываются в качестве функций, а привязку `this` получают только вызовы методов:

```
Old_object.bud = function bud() {
    const that = this;

// lou не может видеть принадлежащий bud параметр this, но lou может видеть
// принадлежащий bud параметр that.

    function lou() {
        do_it_to(that);
    }
    lou();
};
```

Привязка `this` работает только при вызове методов, поэтому вызов:

```
new_object.bud();
```

выполняется успешно, а:

```
const funky = new_object.bud;
funky();
```

нет. Здесь в `funky` содержится ссылка на ту же самую функцию, что и в `new_object.bud`, но `funky` вызывается как функция, поэтому не имеет привязки `this`.

Привязка `this` характерна тем, что имеет динамическую природу. Все остальные переменные имеют статическую привязку. И это создает путаницу.

```
function pubsub() {

// Фабрика pubsub создает объект публикации-подписки. Код, который
// получает доступ к объекту, может подписаться на функцию, получающую
// публикации, и опубликовать материал, который будет доставлен всем
// подписчикам.

// Для содержания функции subscriber используется массив подписчиков. Он
// скрыт от внешнего мира, поскольку находится в области видимости функции
// pubsub.

    const subscribers = [];
    return {
        subscribe: function (subscriber) {
            subscribers.push(subscriber);
        },
        publish: function (publication) {
            const length = subscribers.length;
            for (let i = 0; i < length; i += 1) {
                subscribers[i](publication);
            }
        }
    };
}
```

Функция `subscriber` получает привязку `this` к массиву `subscribers`, потому что:

```
subscribers[i](publication);
```

является вызовом метода. Он не обязательно похож на вызов метода, но суть от этого не меняется. Это дает каждой функции `subscriber` доступ к массиву `subscribers`, что может позволить `subscriber` нанести вред, например удалить всех остальных подписчиков или перехватить и изменить их сообщения.

```
my_pubsub.subscribe(function (publication) {
    this.length = 0;
});
```

Привязка `this` способна угрожать безопасности и надежности. Когда функция сохраняется в массиве, а позже вызывается, ей дается `this`, привязанная к массиву. Если наличие доступа к массиву для функций не предполагалось, как обычно и бывает, возникает вероятность неблагоприятного развития событий.

Функция `publish` может быть исправлена путем замены цикла `for` циклом `forEach`:

```
publish: function (publication) {
    subscribers.forEach(function (subscriber) {
        subscriber(publication);
    }
}
```

Все переменные статически привязаны, и это хорошо. Динамическая привязка есть только у `this`. Это означает, что ее привязка определяется вызывающей функцией, а не создателем функции. Данная аномалия становится источником путаницы.

У функционального объекта есть два прототипных свойства. В нем имеется делегационная ссылка на `Function.prototype`. Также у него есть свойство `prototype`, содержащее ссылку на объект, используемый в качестве прототипа объектов, созданных функцией, вызванной с префиксом `new`.

Вызов конструктора записывается путем помещения префикса `new` перед вызовом функции. Наличие префикса `new` приводит к следующим действиям:

- созданию значения `this` с `Object.create(`*`функция`*`.prototype)`;
- вызову *`функции`* с `this`, привязанной к новому объекту;
- принудительному возвращению `this`, если *`функция`* не возвращает объект.

Используя функцию `Object.assign` для копирования методов из одного прототипа в другой, можно организовать что-то вроде наследования. Но куда прочнее в обиход вошла замена свойства `prototype` функционального объекта объектом, созданным другим конструктором.

Поскольку каждая функция является потенциальным конструктором, узнать в нужный момент, следует ли в вызове применять префикс `new`, довольно трудно. Хуже того, когда он требуется, но по забывчивости не поставлен, никаких предупреждений не выдается.

Поэтому мы руководствуемся следующим соглашением: функции, предназначенные для вызова в качестве конструкторов с префиксом `new`, должны получать имена, начинающиеся с прописной буквы.

В JavaScript имеется также более привычный, похожий на классы синтаксис, созданный специально для тех разработчиков, которые не знают и никогда не узнают, как работает JavaScript. Он позволяет задействовать навыки работы с менее востребованными языками без обучения.

Синтаксис классов, несмотря на свой внешний вид, не реализует никакие классы. Это всего лишь «синтаксический сахар» в дополнение к странностям конструктора псевдоклассов. Он сохраняет самый худший аспект классической модели — расширения `extends`, создающие сильные сцепления между классами. Сильное сцепление становится причиной создания ненадежных и дефектных конструкций.

Избавление от this

В 2007 году было реализовано несколько исследовательских проектов, в рамках которых предпринимались попытки разработать безопасный поднабор JavaScript. Одной из важнейших проблем было управление привязкой `this`. В вызове метода `this` привязывается к интересующему объекту, что иногда полезно. Но, когда

этот же самый функциональный объект вызывается в качестве функции, `this` может быть привязана к глобальному объекту, что было крайне нежелательно.

Мое решение этой проблемы — полный запрет `this`. Аргументы: проблематичность и ненужность этой привязки. Если мы уберем `this` из языка, его полноценность по Тьюрингу не пострадает. Поэтому я начал программировать в диалекте, свободном от `this`, чтобы выяснить степень трудностей, связанных с отказом от нее.

Как же я удивился тому, что ситуация не усложнилась, а упростилась и мои программы стали меньше и качественнее.

Поэтому я рекомендую убрать `this`. Вы станете более квалифицированным и счастливым программистом, научившись работать без `this`. Я ничего у вас не отнимаю, а предлагаю путь к лучшей жизни через написание более качественных программ.

Программисты, использующие классы, так и покинут этот мир, не узнав, насколько они были несчастны. Привязка `this` не дает ничего хорошего.

Само слово *this* — это указательное местоимение. Наличие *this* в языке затрудняет разговор на нем. Он становится похож на программирование с участием пары знаменитых комиков — Эбботта и Костелло.

Глава 17

Как работает код без классов

●○○○●

> И думаешь, что ты умен вне всяких классов
> и свободен.
>
> *Джон Леннон (John Lennon)*

Одной из ключевых идей в разработке объектно-ориентированного программирования была модель обмена данными между частями программы. Имя метода и его аргументы нужно представлять в виде сообщений. Вызов метода посылает объекту сообщение. Каждый объект характеризуется собственным поведением, которое проявляется при получении конкретных сообщений. Отправитель полагает, что получатель знает, что ему делать с сообщением.

Одной из дополнительных выгод является полиморфизм. Каждый объект, распознавший конкретное сообщение, имеет право на его получение. Что происходит потом, зависит от специализации объекта. И это весьма продуктивная мысль.

К сожалению, мы стали отвлекаться на наследование — весьма эффективную схему повторного использования кода. Его важность связана с возможностью уменьшить трудозатраты при разработке программы. Наследование выстраивается на схожем замысле, за исключением *некоторых нюансов*. Можно сказать, что некоторый объект или класс объектов подобен какому-то другому объекту или классу объектов, но имеет некоторые важные отличия. В простой ситуации все работает замечательно. Следует напомнить, что современное ООП началось со Smalltalk — языка программирования для детей. По мере усложнения ситуации наследование становится проблематичным. Оно порождает сильное сцепление классов. Изменение одного класса может вызвать сбой в тех классах, которые от него зависят. Модули из классов получаются просто никудышными.

Кроме того, мы наблюдаем повышенное внимание к свойствам, а не к объектам. Особое внимание уделяется методам получения (get-методам) и присваивания (set-методам) значений каждому отдельно взятому свойству, а в еще менее удачных проектах свойства являются открытыми и могут быть изменены без ведома объекта. Вполне возможно ввести в обиход более удачный проект, где свойства скрыты, а методы обрабатывают транзакции, не занимаясь только лишь изменением свойств. Но такой подход применяется нечасто.

Кроме того, существует слишком сильная зависимость от типов. Типы стали особенностью Фортрана и более поздних языков, так как были удобны создателям компилятора. С тех времен мифология вокруг типов разрослась, обзаведясь экстравагантными заявлениями о том, что типы защищают программу от ошибок. Несмотря на преданность типам, ошибки не ушли из повседневной практики.

Типы вызывают уважение, их хвалят за раннее обнаружение просчетов на стадии компиляции. Чем раньше будет обнаружена оплошность, тем меньших затрат потребует ее ликвидация. Но при надлежащем тестировании программы все эти просчеты обнаруживаются очень быстро. Поэтому ошибки идентификации типов относятся к категории малозатратных.

Типы не виноваты в появлении труднообнаруживаемых и дорогостоящих ошибок. Их вины нет и в возникновении проблем, вызываемых такими ошибками и требующих каких-то уловок. Типы могут подтолкнуть нас к использованию малопонятных, запутанных и сомнительных методов программирования.

Типы похожи на диету для похудания. Диету не обвиняют в возвращении и увеличении веса. Ее также не считают причиной страданий или вызванных ею проблем со здоровьем. Диеты вселяют надежду, что вес придет в здоровую норму и мы продолжим есть нездоровую пищу.

Классическое наследование позволяет думать, что мы создаем качественные программы, в то время как мы допускаем все больше ошибок и применяем все больше неработоспособных наследований. Если игнорировать негативные проявления, типы представляются крупной победой. Преимущества налицо. Но если присмотреться к типам более пристально, можно заметить, что затраты превышают выгоду.

Конструктор

В главе 13 мы работали с фабриками — функциями, возвращающими функции. Что-то похожее теперь можем сделать с конструкторами — функциями, возвращающими объекты, которые содержат функции.

Начнем с создания `counter_constructor`, похожего на генератор `counter`. У него два метода, `up` и `down`:

```
function counter_constructor() {
    let counter = 0;

    function up() {
        counter += 1;
        return counter;
    }

    function down() {
        counter -= 1;
```

```
        return counter;
    }

    return Object.freeze({
        up,
        down
    });
}
```

Возвращаемый объект заморожен. Он не может быть испорчен или поврежден. У объекта есть состояние. Переменная `counter` — закрытое свойство объекта. Обратиться к ней можно только через методы. И нам не нужно использовать `this`.

Это весьма важное обстоятельство. Интерфейсом объекта являются исключительно методы. У него очень крепкая оболочка. Мы получаем наилучшую инкапсуляцию. Прямого доступа к данным нет. Это весьма качественная модульная конструкция.

Конструктор — это функция, возвращающая объект. Параметры и переменные конструктора становятся закрытыми свойствами объекта. В нем нет открытых свойств, состоящих из данных. Внутренние функции становятся методами объекта. Они превращают свойства в закрытые. Методы, попадающие в замороженный объект, являются открытыми.

В методах должны реализовываться транзакции. Предположим, к примеру, что у нас есть объект `person`. Может потребоваться изменить адрес того лица, чьи данные в нем хранятся. Для этого не нужен отдельный набор функций для изменения каждого отдельного элемента адреса. Нужен один метод, получающий объектный литерал, способный дать описание всех частей адреса, нуждающихся в изменении.

Одна из блестящих идей в JavaScript — объектный литерал. Это приятный и выразительный синтаксис для кластеризации информации. Создавая методы, потребляющие и создающие объекты данных, можно сократить количество методов, повышая тем самым целостность объекта.

Получается, у нас есть два типа объектов.

- Жесткие объекты содержат только методы. Эти объекты защищают целостность данных, содержащихся в замыкании. Они обеспечивают нас полиморфизмом и инкапсуляцией.
- Мягкие объекты данных содержат только данные. Поведение у них отсутствует. Это просто удобная коллекция, с которой могут работать функции.

Есть мнение, что ООП началось с добавления процедур к записям языка Кобол, чтобы обеспечить тем самым некое поведение. Полагаю, что сочетание методов и свойств данных было важным шагом вперед, но стать последним шагом не должно.

Если жесткий объект должен быть преобразован в строку, нужно включить метод `toJSON`. Иначе `JSON.stringify` увидит его как пустой объект, проигнорировав методы и скрытые данные (см. главу 22).