JSDG 🦏 | Classes | Классы | chapter 9

`JavaScript: The Definitive Guide 7th EDITION • Master the World's Most-Used Programming Language • David Flanagan • 2021`

___

Однако часто бывает полезно определять класс объектов, которые разделяют
определенные свойства.

Члены, или экземпляры, класса имеют собственные свойства для хранения или определения их состояния, но они также располагают методами, которые устанавливают их поведение.

Такие методы определяются классом и разделяются всеми экземплярами.

Например, вообразим себе класс по имени Complex, который представляет и выполняет арифметические действия с комплексными числами.

Экземпляр Complex мог бы иметь свойства для хранения действительной и мнимой частей (состояния) комплексного числа. Вдобавок в классе Complex были бы определены методы для выполнения сложения и умножения (поведение) комплексных чисел.

Классы в JavaScript используют наследование, основанное на прототипах:
если два объекта наследуют свойства (как правило, свойства-функции, или методы) из одного и того же прототипа, то мы говорим, что эти объекты являются экземплярами того же самого класса. Так в двух словах работают классы
JavaScript.

В ES6 появился совершенно новый синтаксис (в том числе ключевое слово class) , который еще больше облегчает создание классов.

___

Классы и прототипы

Класс в JavaScript представляет собой набор объектов, которые наследуют свойства от того же самого объекта прототипа.

```
// фабричная функция, которая возвращает новый объект диапазона.
function range (from, to) 1

    // Использовать Object.create () для создания объекта, который наследуется
    // от объекта прототипа, определенного ниже. Объект прототипа хранится как
    // свойство этой функции и определяет разделяемые методы (поведение)
    // для всех объектов, представляющих диапазоны.

    let r = Object. create (range. methods) ;

    // Сохранить начальную и конечную точки (состояние) нового объекта диапазона.
    // Эти свойства не являются унаследованными и они уникальны для этого объекта.

    r.from = from;
    r.to = to;

    // В заключение возвратить новый объект.
    return r;
}

// объект прототипа, определяющий свойства, которые наследуются
// всеми объектами, представляющими диапазоны.
range.methods = {
    // Возвращает true, если x входит в диапазон, и false в противном случае.
    // Метод работает как с числовыми, так и с текстовыми диапазонами
    // и диапазонами Date.
```

```
    includes (x) { return this.from <= x && x <= this.tom),

    // Генераторная функция, которая делает экземпляры класса
итерируемыми.
    // Обратите внимание, что она работает только с числовыми
диапазонами.

    *(Symbol.iterator] () {
        for (let x = Math.ceil (this.from); x <= this.to; x++) yield x;
    },

    // Возвращает строковое представление диапазона.
    toString () { return "(" + this.from + " "..."+ this.to + ")"; }
};

// Пример использования объекта диапазона.
let r = range (1, 3); // Создать объект диапазона
r.includes (2) // => true: 2 входит в диапазон
r.toString () // => " (1...3) "
[...r] // => [1, 2, 3]: преобразовать в массив через итератор
```

Еще одно критически важное отличие между примерами 9.1 и 9.2 связано со способом именования объекта прототипа. В первом примере прототипом был range .methods.

Это удобное и описательное имя, но оно произвольно. Во вто ром примере прототипом является Range.prototype, и такое имя обязатель но. Вызов конструктора Range () приводит к автоматическому использованию Range.prototype в качестве прототипа нового объекта Range.

Конструкторы, идентичность классов и операция instanceof

Хотя конструкторы не настолько основополагающие как прототипы, конструктор служит публичным лицом класса. Наиболее очевидно то, что имя конструктора обычно принимается как имя класса. Например, мы говорим, что конструктор Range () создает объекты Range. Однако по

существу конструкторы применяются в качестве правостороннего операнда операции instanceof при проверке объектов на предмет членства в классе. Если мы имеем объект r и хотим выяснить, является ли он объектом Range, то можем записать:

r instanceof Range / => true: r наследуется от Range.prototype

Говоря формально, в предыдущем примере кода операция instanceof не проверяет, действительно ли объект r был инициализирован конструктором Range. Взамен она проверяет, унаследован ли объект г от Range. prototype.

Если мы определим функцию Strange () и установим ее прототип таким же, как Range.prototype, тогда объекты, создаваемые с помощью new Strange (), будут считаться объектами Range с точки зрения операции instanceof (тем не менее, на самом деле они не будут работать как объекты Range, потому что их свойства from и to не инициализированы):

```
function Strange () {}
Strange.prototype = Range.prototype;

new Strange() instanceof Range // => true
```

Если вы хотите проверить цепочку прототипов объекта на предмет наличия специфического прототипа и не хотите применять функцию конструктора в качестве посредника, тогда можете использовать метод isPrototypeof ().

Скажем, в примере 9.1 мы определяли класс без функции конструктора, а потому применить instanceof с таким классом не удастся. Однако взамен мымогли бы проверить, был ли объект r членом такого класса без конструктора, посредством следующего кода:

```
range.methods.isPrototypeof(r); // range.methods - объект прототипа
```

Свойство constructor

В качестве конструктора можно использовать любую функцию JavaScript (кроме стрелочных, генераторных и асинхронных функций), а для вызова функции как конструктора нужно лишь свойство prototype.

```
let F = function () {}; // Объект функции
let p = F.prototype; // Объект прототипа, ассоциированный с F
let c = p.constructor; // Функция, ассоциированная с прототипом
C === F // => true: F.prototype.constructor === F для любого объекта F

let o = new F() ; // Создать объект o класса F
o.constructor === F // => true: свойство constructor указывает класс
```

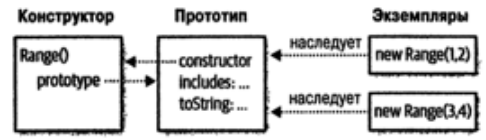

Рис. 9.1. Функция конструктора, ее прототип и экземпляры

—

Классы с ключевым словом class

```
class Range {
    constructor (from, to) (
        // Сохранить начальную и конечную точки (состояние)
        // нового объекта диапазона.

        // Эти свойства не являются унаследованными и они уникальны
        // для этого объекта.
        this.from = from;
        this.to = to;
```

```
    }

    // Возвращает true, если x входит в диапазон, и false в
противном
    // случае. Метод работает как с числовыми, так и с
текстовыми
    // диапазонами и диапазонами Date.

    includes (x) { return this.from <= x && x <= this.to; }

    // Генераторная функция, которая делает экземпляры класса
итерируемыми.
    // обратите внимание, что она работает только с числовыми
диапазонами.
    * [Symbol.iterator] () {
        for (let x = Math.ceil (this.from); x <= this.to; x++) yield x;
    }

    // Возвращает строковое представление диапазона.
    toString () { return `($(this.from)...$(this.to))`; }
}

// Пример использования нового класса Range.
let r = new Range (1, 3); // Создать объект Range
r.includes(2) // => true: 2 входит в диапазон
r.toString () // => " (1...3)"
[...r] // => [1, 2, 3]: преобразовать в массив через итератор
```

Ключевое слово constructor используется для определения функции конструктора класса.

Тем не менее, определяемая функция на самом деле не называется "constructor". Оператор объявления class определяет новую переменную Range и присваивает ей значение этой специальной функции
constructor.

Если ваш класс не нуждается в инициализации, тогда можете опустить ключевое слово constructor вместе с его телом, и неявно будет создана пустая функция конструктора.

Если вы хотите определить класс, который является подклассом другого класса, или унаследованным от него, то можете применить вместе с class ключевое слово extends

```
// Класс Span подобен Range, но вместо инициализации значениями начала и конца мы инициализируем его значениями начала и длины.
class Span extends Range I
	constructor (start, length) {
		if (length >= 0) {
			super (start, start + length);
		} else {
			super (start + length, start);
		}
	}
}
```

Создание подклассов - отдельная тема. Мы возвратимся к ней в разделе 9.5 и обсудим ключевые слова extends и super.

___

Статические методы

Определить статический метод внутри тела класса можно, снабдив объявление метода префиксом в виде ключевого слова static.

Статические методы определяются как свойства функции конструктора, а не свойства объекта прототипа.

Иногда статические методы называют методами класса из-за того, что они вызываются с применением имени класса/конструктора. Такой термин используется как противоположность обыкновенным методам экземпляра, которые вызываются на экземплярах класса. Поскольку статические методы вызываются на конструкторе, а не на индивидуальном экземпляре, в них почти никогда не имеет смысла применять ключевое слово this.

___

генераторныи метод с вычисляемым именем, который делает класс Range итерируемым:

```
* [Symbol.iterator] () {
    for (let x = Math. ceil (this.from); x <= this.to; x++) yield x;
}
```

___

Открытые, закрытые и статические поля

При обсуждении классов, определяемых с помощью ключевого слова class, мы затронули только определение методов внутри тела класса. Стандарт ES6 разрешает создавать только методы (включая методы получения, установки и генераторы) и статические методы; синтаксис для определения полей отсутствует.

Однако стандартизация для расширенного синтаксиса, который делает возможным определение полей экземпляра и статических полей в открытой и закрытой форме, уже ведется.

```
class Buffer {
    constructor {
        this.size = 0;
        this.capacity = 4096;
        this.buffer = new Uint8Array (this.capacity);
    }
}

class Buffer {
    size = 0;
    capacity = 4096;
    buffer = new Uint8Array (this.capacity);
}
```

поле, имя которого начинается с # (обычно недопустимый символ в идентификаторах JavaScript), то такое поле будет пригодным к употреблению (с префиксом #)

Если для предшествующего гипотетического класса Buffer вы хотите гарантировать, что пользователи не сумеют случайно модифицировать поле size экземпляра, тогда могли бы взамен использовать закрытое поле #size и определить функцию получения, чтобы предоставить доступ только для чтения к значению:

```
class Buffer {
      #size = 0;
      get size ()  { return this.#size; }
}
```

Обратите внимание, что закрытые поля должны объявляться с применением нового синтаксиса полей до того, как их можно будет использовать. Вы не можете просто записать this.#size = 0; в конструкторе класса, если только не включили "объявление" поля прямо в тело класса.

Подклассы

В объектно-ориентированном программировании класс B может *расширять* или *быть подклассом* класса A.

Мы говорим, что A является суперклассом, а B — подклассом

Существует несколько важных правил, которые нужно знать об использовании **super () в** конструкторах.

• Если вы определяете класс с ключевым словом extends, тогда в конструкторе вашего класса должно применяться super () для вызова конструктора суперкласса.

• Если вы не определили конструктор в своем подклассе, то он будет определен автоматически. Такой неявно определенный конструктор просто передает super () любые предоставленные значения.

• Вы не можете использовать ключевое слово this в своем конструкторе, пока не будет вызван конструктор суперкласса

посредством super(). Это обеспечивает соблюдение правила о том, что суперклассы должны инициализировать себя перед подклассами.

• Специальное выражение new. target не определено в функциях, которые вызываются без ключевого слова new. Тем не менее, в функциях конструкторов new.target является ссылкой на вызванный конструктор. Когда конструктор подкласса вызывается и применяет super () для вызова конструктора суперкласса, то конструктор суперкласса будет видеть конструктор подкласса как значение new. target. Хорошо спроектированный суперкласс не должен знать, создаются ли из него подклассы, но new.target. name иногда полезно использовать, например, в журнальных сообщениях.

Иерархии классов и абстрактные классы

Применение классов JavaScript для инкапсуляции данных и модульной организации кода часто является великолепным приемом, и вы можете обнаружить, что регулярно используете ключевое слово class.

Но может выясниться, что вы отдаете предпочтение композиции перед наследованием и редко нуждаетесь в применении extends (кроме случаев, когда вы используете библиотеку или фреймворк, который требует расширения своих базовых классов).

В примере 9.8 определено много подклассов, но также показано, как можно определять абстрактный класс, т.е. класс, не включающий полную реализацию, который послужит общим суперклассом для группы связанных подклассов.

Абстрактный суперкласс может определять частичную реализацию, которую наследуют и разделяют все подклассы. Затем подклассам останется лишь определить собственное уникальное поведение, реализовав абстрактные методы, которые определены, но не реализованы суперклассом.

Обратите внимание, что в JavaScript отсутствует формальное

определение абстрактных методов или абстрактных классов;
я просто применяю здесь такое название для нереализованных методов и классов, реализованных не полностью.

abstract class.pdf
PDF-документ · 1,1 МБ