

Курс: Javascript. Событийно-ориентированное программирование

Тема 6. Создание объектов.

Литеральный объект (plain JavaScript object) представляет собой обычный объект, ссылочный тип данных. В современном JavaScript согласно ES2015 (раздел 4.3) существуют 4 категории объектов:

1. Обычные (ordinary) объекты (обладают всеми чертами поведения объектов, которые поддерживаются в JavaScript).
2. Необычные (exotic) объекты (обладают чертами поведения, отличающими их от обычных объектов).
3. Стандартные (standard) объекты (определяются в ES2015, такие как Array, Date, Map и т. д., могут быть обычными и необычными).
4. Встроенные (built-in) объекты (определяются средой выполнения JavaScript, в которой действует сценарий; все стандартные объекты одновременно являются встроенными).

Например, объекты, которые создаются с помощью конструктора Array — необычные. Их свойство length отслеживает и способно изменять элементы массива.

Синтаксис литерального объекта напоминает синтаксис блока. Это одна или более пар «ключ»:«значение», представленных в виде списка через запятую. В «родном» синтаксисе допускается последняя запятая (trailing comma), а в JSON нет. Правило comma-dangle линтера eslint подробно описывает различные установки, при которых считаются допустимыми те или иные варианты постановки запятой, по аналогии с списком аргументов функции, списком элементов в массиве.

Литеральный объект напоминает тип-запись в Паскале или структуру в C, но близок и так называемым ассоциативным массивам. В ES2015 ассоциативные массивы введены как самостоятельный тип Map.

Простейший способ создать литеральный объект для дальнейшей работы с ним — присваивание, в котором слева находится переменная, а справа сам объект:

```
const person = { first: 'Elias', last: 'Goss' };  
               оператор уточнения      безымянный литерал объекта  
console.log(person.first); // Elias
```

Литерал объекта и уточнение

При дотошном следовании стилю кодирования возникает вопрос о «правильном» написании объекта. Такие правила линтера, как object-curly-spacing, key-spacing, comma-spacing определяют этот стиль:

```
• const person = {first: 'Elias', last: 'Goss'};
```

A space is required after '{'. (object-curly-spacing)

Использование пробела в синтаксисе объектов

В «родном синтаксисе» ключи могут быть написаны «как есть», без кавычек, для чего они должны быть валидными идентификаторами. В JSON ключи должны заключаться, как и строковые значения, в двойные кавычки. Кроме того, значениями в литералах могут быть, например, функции, а в JSON мы ограничены строками, числами, массивами и другими объектами.

Сериализация позволяет перейти от «родного» формата объектов JavaScript к формату JSON. При этом происходит усечение его возможностей. Например, не может быть функций в качестве значений, кавычки становятся только двойными и т.п. Для преобразования мы используем метод `JSON.stringify`: например `JSON.stringify(person)`. Обратно, чтобы получить объект по строке, содержащей валидный JSON, мы используем метод `JSON.parse`.

Ключи также называют свойствами, но эти термины употребляются обычно в разных контекстах. Когда речь идёт о внутреннем устройстве объекта, с позиции разработчика объекта, говорят о ключах. Когда речь идёт об обращении к объекту извне, с позиции пользователя объекта, говорят о свойствах.

Если ключ не является валидным идентификатором, например, будет содержать пробел, то его нельзя использовать с оператором уточнения, но можно использовать с оператором индексации.

Ключ литерального объекта представляет собой переменную внутри объекта. Это имя можно сохранить в другой переменной, в виде строки. Тогда его можно использовать в объекте опосредованно через эту другую переменную, указав её имя в квадратных скобках, что называется вычисляемыми ключами или свойствами.

```
const first = 'firstName';  
const person = { [first]: 'Elias', last: 'Goss' };  
console.log(person.firstName); // Elias
```

Но оказывается, что вычисляемые ключи могут делать более интересные вещи. Рассмотрим следующий пример.

```
const  
  first = x => x,  
  second = first;  
const person = { [first]: 'Elias', last: 'Goss' };  
console.log(person[second]); // Elias
```

В первых трёх строках этого примера с именем `first` связывается функция, а имя `second` становится синонимом имени `first`, т.е. указывает на ту же область памяти, где хранится код функции `x => x`.

Таким образом, мы можем зафиксировать некоторое значение (в данном случае это функция) и далее оперировать не им самим, а ссылками на него. Развитие этой идеи привело к появлению особого типа данных Symbol, представители которого доступны только в виде ссылок.

Что *реально* является ключом, значением которого является строка 'Elias'?

Ответ: то значение, которое доступно как first.toString()

Т.е. ключом является представление значения, связанного с именем first, в виде строки.

```
const first = x => x;
const person = { [first]: 'Elias', last: 'Goss' };

console.log(person[Object.keys(person)[0]]); // Elias
console.log(person[first.toString()]); // Elias
console.log(person['x => x']); // Elias
console.log(person[x => x]); // Elias
```

Последний вариант обращения может показаться странным. Может показаться, что мы используем в качестве ключа функцию. Но это не так. Ведь если бы это было так, то в последней строке мы бы имели дело с другой функцией (вновь созданным литералом), а не с той же самой, которая фигурирует в первой строке, хотя выглядят они совершенно одинаково. Просто интерпретатор тут же преобразует «литерал функции» в строковое представление, и оно оказывается совпавшим с тем, которое хранится в качестве ключа объекта person.

Понятно, что обращение через точку в таких случаях невозможно. А обращение person.first даст нам undefined, т.к. ключа 'first' не существует.

Наименование «объект» может, вообще говоря, быть несколько дезориентирующим, поскольку в классических объектно-ориентированных языках объект есть экземпляр класса. Так, в PHP чтобы получить объект, у которого есть свойства и методы, нужно объявить класс и экземплифицировать его. Это создание объектов от общего к частному.

Когда объект создаётся литералом, то на самом деле создаётся ещё один объект, связанный с ним, который называется прототипом. Это создание объектов от частного ... к другому частному или, возможно, к общему — если необходимо, можно создать новый объект на основе уже существующего, используя уже существующий объект как прототип.

Чтобы создать «схему генерации объектов», в традиционном JavaScript используется функция-конструктор, которая, как правило, заполняет значения ключей объекта, а отдельно методами заполняется прототип будущих объектов, доступный через свойство prototype конструктора; с некоторыми ухищрениями можно реализовать наследование; классом *в этом случае* является множество объектов, созданных с помощью этого конструктора; в ES2015 мы получили возможность объявлять классы и создавать объекты с помощью классов, хотя «под капотом» используется механизм прототипов.

Литеральные объекты — это в некотором смысле одиночки, они не предполагают своего «размножения» в явной форме, создают своего рода пространство имён и изолируют свои переменные от внешнего мира. В этом на них похожи блоки, введённые в ES2015. Но блоки

изолируют свои переменные абсолютно — обратиться к переменным внутри блока снаружи невозможно, — а к переменным (ключам) объекта обратиться можно.

```
const sayLast = ({ last }) => last;  
console.log(sayLast({ first: 'Elias', last: 'Goss' })); // Goss
```

В этом примере мы передаём безымянный объект с двумя ключами и двумя значениями в функцию. Функция использует паттерн деструктуризации объекта, извлекая из него значение по ключу, указанному в фигурных скобках, и возвращая его. Т.е. это функция извлечения значения по ключу. У её формального параметра нет явного имени, обращение к переданному объекту происходит неявно.


Изменим немного вышеприведённый пример. Что делает эта функция в модифицированном виде?

```
const sayLast = ({ last }) => ({ last });  
console.log(sayLast({ first: 'Elias', last: 'Goss' })); // ?
```

Ответ: она формирует новый объект, у которого будет только один ключ (last), значением которого будет значение, доступное по имени last благодаря работе деструктуризатора в её сигнатуре. В данном случае этим значением будет 'Goss'.

В правой части функции используется сокращённый синтаксис создания литеральных объектов, который можно пояснить таким изображением:

```
const sayLast = ({ last }) => ({ last: last });
```



Объяснение сокращённого синтаксиса объекта

Чтобы обратиться к значению внутри объекта, необязательно, чтобы объект был ассоциирован с каким-либо именем:

```
console.log({ first: 'Elias', last: 'Goss' }.last); // Goss  
console.log(Object.values({ first: 'Elias', last: 'Goss' })[1]); // Goss  
console.log(Reflect.get({ first: 'Elias', last: 'Goss' }, 'last')); // Goss
```

С помощью объектов мы можем создавать структуры, напоминающие списки или очереди, в которых каждый элемент имеет ссылку на последующий или предыдущий. Рассмотрим, как с помощью цикла for и коллбэка мы можем осуществить перебор такой структуры:

```
const o5 = { name: 'Obj5' };  
const o4 = { name: 'Obj4', next: o5 };  
const o3 = { name: 'Obj3', next: o4 };  
const o2 = { name: 'Obj2', next: o3 };  
const o1 = { name: 'Obj1', next: o2 };  
function tail(o, whatToDo) {  
  for (; o.next; o = o.next) whatToDo(o);  
  return o;  
}  
const res = tail(o1, x => console.log(x.name)).name;
```

```
console.log(res);
```

Передавая функции объект вместо аргументов через запятую, мы делаем сигнатуру функции более читаемой, реализуем возможность управлять именами и порядком аргументов. Например, встроенная в JavaScript функция `defineProperty` (метод класса `Reflect`), определяет новое свойство объекта с помощью конфигурационного объекта, в котором есть такие ключи как `value` и `enumerable`:

```
const person = { firstName: 'Elias' };
Reflect.defineProperty(person, 'lastName', {
  value: 'Goss', enumerable: true
});
```

Сформировался даже паттерн `RO[RO]` (`Receive Object[Return Object]`), подразумевающий передачу функции параметров в виде конфигурационного одиночного объекта (и, возможно, возвращать данные в аналогичной форме). Сочетая передачу объектов с деструктуризацией, мы можем добиваться элегантного и компактного кода:

```
// RORO
const sayName = function ({ first: f, last: l = 'Goss' } = {}) {
  return `${f} ${l}`;
};
console.log(sayName({ first: 'Elias' })); // Elias Goss
```

Когда мы передаём функции объект с целью сгруппировать несколько аргументов (в том числе это касается функций-конструкторов), то для краткости можем говорить, например «объект "methods"», и под этим подразумевается объект, который является значением ключа "methods", который, в свою очередь, сам принадлежит какому-то объекту. Вот пример применения указанной выше ситуации:

```
new Vue({
  el: '#app',
  data: {
    title: 'Приложение Vue',
    name: 'Elias'
  },
  methods: {
    sayHello(time) {
      return `Hello and good ${time}, ${this.name}!`;
    }
  }
});
```

В фреймворке `Vue` мы создаём экземпляр объекта с помощью конструктора `Vue` и передаём этому конструктору объект конфигурации, который включает.

Методы

Каким образом создаются *методы*?

```
const person = {
  first: 'Elias',
  sayHello() {
    return `Hello, ${this.first}!`;
  }
};
console.log(person.sayHello()); // Hello, Elias!
```

Простейший вариант включения функции внутрь объекта показан выше. Внутри объекта слово `this` указывает на этот объект. Но это касается только стандартно объявленных функций (слово `function` в примере отсутствует вместе с двоеточием в силу сокращённого синтаксиса!), а у стрелочных функций нет такой возможности.

Поскольку ключи создаются простым обращением к ним (уточнением или индексацией), то создать метод можно присвоив ключу функцию (именованную, безымянную, стрелочную).

Предположим, что у нас есть два объекта: `person1` и `person2`, в первом из которых есть метод `sayHello`, а во втором нет. Может ли `person1` одолжить свой метод объекту `person2`?

Да, это возможно благодаря таким методам как `call`. Первым аргументом мы передаём контекст, который доступен как `this`.

```
const person1 = {
  first: 'Elias',
  sayHello() {
    return `Hello, ${this.first}!`;
  }
};
const person2 = { first: 'Victor' };
console.log(person1.sayHello()); // Hello, Elias!
console.log(person1.sayHello.call(person2)); // Hello, Victor!
```

Менеджмент ключей (свойств и методов) имеет свои тонкости, которые связаны с:

- подходом к созданию (литералы или классы);
- метасвойствами (атрибутами).

При выводе объекта (с помощью `valueOf` или сериализацией), а также при использовании таких методов как `Object.keys`, видны не все ключи (не обязаны быть видны), которые становятся видны при более детальном рассмотрении. С другой стороны, при переборе с помощью цикла `for..in` в число ключей попадают те, которые принадлежат к цепочке прототипов.

Энумераторность и итераторность

Ключи и значения соответствуют именам и сущностям .
--

<p>Энумерабельность: доступны имена, а через них - сущности (литеральный объект, энумерабелен и не итерируем, есть for..in нет for..of)</p> <p>задаётся метасвойством (атрибутом ключа)</p>	<p>Итерируемость: доступны сущности, и не через имена, а путём продвижения/обхода (обычный массив: энумерабелен и итерируем есть и for..in и for..of)</p> <p>задаётся специальным методом Symbol.iterator</p>
--	--

Массивоподобные объекты

В некоторых случаях объекты, которые не созданы литералом массива или явно с помощью конструктора Array, напоминают по своему поведению массивы. По-английски они называются array-like objects (ALO). Рассмотрим пример:

https://kodaktor.ru/events_alo

У объекта, обозначенного как t, есть метод forEach и свойство length. Тем не менее, при попытке изменить значение его свойства length с ним не происходит никаких изменений. Кроме того, отличие от массива состоит в том, что его свойство length доступно циклу for..in. Этот объект, коллекция узлов, реализует интерфейс NodeList, и не создавался с помощью конструктора Array. Тем не менее, его можно итерировать. В новых версиях реализации DOM появилась поддержка метода forEach и встроенный итератор для цикла for..of.

Прототипы и классы

При создании литерального объекта автоматически создаётся ещё один объект, который называется прототипом. Но это касается всех сущностей, которые относятся к объектам. Более того, это относится и к примитивным типам данных.

Например, когда мы создаём переменную, которой присваиваем строковое значение, мы можем сразу обратиться к её свойству length или методу includes. Но свойства есть только у объектов. Но при обращении к строке как к объекту происходит создание объекта-обёртки. И у этого объекта есть прототип, содержащий метод includes:

```
const first = 'Elias';
first.includes('li'); // true
first.includes('ea'); // false
// вызов через прототип
String.prototype.includes.call(first, 'li'); // true
String.prototype.includes.call(first, 'ea'); // false

Reflect.apply(String.prototype.includes, first, ['li']); // true
Reflect.apply(String.prototype.includes, first, ['ea']); // false
```


Выше показаны три способа обращения к методу `includes`:

- прямой;
- косвенный;
- метапрограммистский.

По способу указания в справочных материалах можно быстро понять, находится ли ключ в прототипе всех объектов данного класса, или только в самом классе:



Запись типа `Object.prototype.propertyIsEnumerable` означает, что вызывать метод `propertyIsEnumerable` нужно уточнением конкретного объекта. Запись типа `Object.getOwnPropertyNames` означает, что вызывать метод нужно ровно как он представлен, передавая ему нужные аргументы (включая конкретный объект).

Объекты обёртки создаются автоматически. Основой для их создания являются конструкторы (`String`, `Number`, `Boolean` — это функции, спроектированные как конструкторы). Всегда можно обратиться к ним в явной форме:

просто вызвав эти функции, передав им в качестве аргумента литерал строки или значение, которое будет преобразовано в строку. Поскольку это именно конструкторы, их можно вызвать с помощью оператора `new` (т.е. экземпляризацией, см. модели вызова функций). Однако конкретно в случае с конструкторами обёрток это считается нереконмендованной практикой (<https://eslint.org/docs/rules/no-new-wrappers>).

Конструктор — это функция, объявленная с помощью слова `function` (не стрелочная), про которую предполагается, что будет основой для создания объектов некоторой одинаковой структуры. Соответственно, чтобы это имело смысл, такая функция должна эту структуру создавать, а следовательно, она должна быть заранее спроектирована в аспекте свойств (и методов).

Вызов конструктора осуществляется с оператором `new`. Если конструктор предполагает наличие аргументов, то вызов сопровождается скобками, но рекомендуется использовать скобки в любом случае (<https://eslint.org/docs/rules/new-parens>).

Функция может определить факт вызова с ключевым словом `new`, и для этого ES2015 определяет метасвойство `new.target`. Метасвойство — это необъектное свойство с дополнительной информацией о его цели (такой, как `new`). Целью обычно является конструктор вновь созданного экземпляра объекта, который будет присвоен ссылке `this` в теле функции. Т.е. `new.target` — это ссылка на функцию, вызванную с ключевым словом `new`. При вызове с помощью `call` или `apply` `new.target` получает значение `undefined`. Метасвойство `new.target` можно также использовать в конструкторах классов, чтобы определить, как вызывался класс. В простейшем случае `new.target` содержит ссылку на функцию-конструктор класса.

Ничто не мешает использовать произвольную `function`-функцию в роли конструктора, но это в целом не особенно полезно:

```
const Cube = function (x) { return x ** 3; };
const cb = new Cube(5);
console.log(cb); // Cube {}
console.log(cb.constructor); // [Function: Cube]
```

Свойство `constructor`, доступное в новом объекте (оно содержится в его прототипе), позволяет определить, какая функция была вызвана как конструктор для этого объекта.

В частности, это может быть анонимная функция:

```
/* eslint new-parens: 0 */
const cb = new function () { return { first: 'Elias' }; };
console.log(cb); // { first: 'Elias' }
console.log(cb.constructor); // [Function: Object]
```

Как видно из примера выше, если конструктор возвращает *объект*, то он и является тем, что возвращает оператор экземпляризации.

Если конструктор не возвращает объект, то создаётся новый объект, который должен был бы заполняться свойствами в рамках конструктора, и в следующем примере он получается пустым:

```
/* eslint new-parens: 0 */
const cb = new function () { };
console.log(cb); // {}
console.log(cb.constructor); // [Function]
```

Каким же образом конструктор должен заполнять структуру вновь создаваемого объекта?

Когда объект создаётся (т.е. когда конструктор не возвращает объект), он доступен внутри конструктора через слово `this`. Он тогда будет тем объектом, который возвращает оператор экземпляризации.

```
const Cube = function (x) { this.x = x; };
const cb = new Cube(5);
console.log(cb); // Cube { x: 5 }
console.log(cb.constructor); // [Function: Cube]
```

Таким образом, у объекта cb теперь есть свойство x с некоторым значением.

Чтобы объект оправдывал своё название объекта, у него должны быть методы.

Методы принято добавлять не в теле конструктора, а отдельно, записывая их в качестве свойств объекта, доступного через свойство prototype функции-конструктора.

```
const Cube = function (x) { this.x = x; };
Cube.prototype.volume = function () { return this.x ** 3; };
const cb = new Cube(5);
console.log(cb); // Cube { x: 5 }
console.log(cb.constructor); // [Function: Cube]
console.log(cb.volume()); // 125
```

Такое обращение с методами объясняется тем, что значения свойств у каждого конкретного объекта предполагаются своими, а методы нет никакой нужды копировать в конкретный объект.

Фактически можно смотреть на прототип как на расшаренный между объектами набор методов, на который у них у всех есть ссылка. Естественно, изменения в прототипе мгновенно отражаются на всех объектах, имеющих этот прототип.

Свойство prototype является особенным. Для обычного объекта оно ничем не выделяется среди всех прочих возможных свойств. Присвоение значения этому свойству для них не играет никакой специфической роли. Иное дело — конструкторы. Array, String и т.п. имеют прототипы, в которых находятся методы, которые можно вызвать в контексте каждого литерала массива, строки и т.п.

При этом следует помнить, что объект-прототип доступен через свойство prototype только в одной ситуации, приведённой выше: когда по конструктору создаётся новый объект. В примере выше метод volume будет находиться в прототипе объекта cb, но не будет доступен через его свойство prototype.

Каким же образом мы можем убедиться в том, что свойство или метод находится в прототипе объекта?

Собственные ключи

Собственным (own) называется ключ (свойство) объекта, который находится в этом объекте, но не в его прототипе. Мы можем получить массив собственных ключей как минимум двумя способами:

- Object.getOwnPropertyNames
- Reflect.ownKeys

Если мы можем написать `cb.constructor` и получаем некоторый результат, но этого свойства нет в массиве собственных ключей, вывод только один: это свойство находится в прототипе. Но в примере выше мы определили только метод `volume`. Откуда же берётся свойство `constructor`?

Когда мы писали `Cube.prototype`, то подразумевали, что объект, скрывающийся за этим выражением, уже существует. Любая функция получает свойство `prototype`, потому что любая функция может быть использована как конструктор. Свойство `prototype` ссылается в этом случае на объект, у которого установлено свойство `constructor`.

```
const Cube = function (x) { this.x = x; };  
console.log(Cube.prototype); // Cube {}  
console.log(Object.getOwnPropertyNames(Cube.prototype)); // [ 'constructor' ]  
console.log(Cube.prototype.constructor); // [Function: Cube]
```

Это свойство `constructor` ссылается на саму функцию, которой принадлежит свойство `prototype`, содержащее это свойство `constructor`.

Мы использовали метод `Object.getOwnPropertyNames` чтобы убедиться, что свойство `constructor` находится в объекте, доступном как `Cube.prototype`. При экземпляризации этот объект станет прототипом экземпляра класса `Cube` и именно поэтому мы сможем получить его значение. Соответственно, это не будет его собственным свойством. Единственным собственным свойством будет `x`.

```
const cb = new Cube(5);  
console.log(cb); // Cube { x: 5 }  
console.log(cb.constructor); // [Function: Cube]  
console.log(Object.getOwnPropertyNames(cb)); // [ 'x' ]
```

Однако почему в предыдущем примере при выводе в консоль `Cube.prototype` мы не увидели там свойства `constructor`?

Ответ: потому что оно обозначено как неперечислимое, а такие свойства выводятся только с помощью специальных методов вроде методов получения собственных ключей.

Чтобы получить доступ к прототипу объекта, мы не можем обратиться к его свойству `prototype`. Свойство `prototype`, если вообще имеет значение, отличное от `undefined`, относится не к объекту, которому принадлежит, а к объектам, создаваемым на основе конструктора. Для получения доступа к прототипу объекта `o`, мы используем:

- Reflect.getPrototypeOf(o)
- o.__proto__

Причем второй способ, хотя и часто работает по факту, но является `deprecated`.

Любой объект может стать прототипом для другого объекта. Для этого можно предложить как минимум три варианта действий:

- присвоить этот объект свойству прототип конструктора
- использовать метод `Object.create`
- использовать метод `Reflect.setPrototypeOf`

Цепочка прототипов

Соответственно существует цепочка прототипов, т.е. когда объект является чьим-то прототипом, и при этом сам имеет прототип.

В JavaScript определена сущность, которая становится «изначальным прототипом», т.е. находится в начале этой цепочки, когда объекты создаются автоматически. Она доступна через выражение `Object.prototype`

```
console.log(Reflect.getPrototypeOf({}) === Object.prototype); // true
const o = Reflect.getPrototypeOf(Object.prototype);
console.log(o); // null
console.log(require('util').format('%o', o)); // 'null'
```

Т.е. у того, что известно как `Object.prototype`, нет и не может быть прототипа.

В нашем примере прототипом прототипа экземпляра класса `Cube` тоже является `Object.prototype`

```
const Cube = function (x) { this.x = x; };
Cube.prototype.volume = function () { return this.x ** 3; };
const cb = new Cube(5);
console.log(cb.constructor); // [Function: Cube]

const cbProt = Reflect.getPrototypeOf(cb); // Cube { volume: [Function] }
console.log(cbProt === Cube.prototype); // true
const cbProtProt = Reflect.getPrototypeOf(cbProt);
console.log(cbProtProt === Object.prototype); // true
```

`Cube` есть конструктор класса `Cube`. Поэтому объект `cb` приобрёл в качестве прототипа то, что находится в `Cube.prototype`.

`Object` есть конструктор класса `Object`. Поэтому, собственно, все объекты, созданные с помощью этого конструктора, и приобретают в качестве прототипа то, что находится в `Object.prototype`.

`Object.prototype` не является единственной сущностью, у которой нет прототипа.

```
const empty = Object.create(null);
const o = Reflect.getPrototypeOf(empty);
console.log(o); // null
```

Возвращаясь к заданному выше вопросу (Каким же образом мы можем убедиться в том, что свойство или метод находится в прототипе объекта?) мы теперь можем дать на него такой ответ:

1. Убедиться, что это свойство или метод не выводится в массиве собственных ключей;
2. Получить прототип и убедиться что это свойство или метод, наоборот, выводится в массиве собственных ключей прототипа (но это может касаться объектов выше в цепочке прототипов вплоть до `Object.prototype`).

Подытоживая, уточним, каково всё-таки взаимоотношение между конструктором и прототипом.

—
Класс — это множество объектов, наследующих свойства от общего объект-прототипа. Это по определению: два объекта являются экземплярами одного класса, только если они наследуют один и тот же прототип.

Но создать несколько объектов, наследующих один и тот же прототип, можно по-разному: фабрично или с помощью конструктора. Фабричное создание объекта не оперирует словом `new` и, следовательно, не задействует конструктор.

Если два объекта наследуют один и тот же прототип, обычно (но не обязательно) это означает, что они были созданы и инициализированы с помощью одного конструктора.

Прототип первичен потому, что

- можно объявить несколько функций и присвоить их свойству `prototype` один и тот же объект, тогда при их вызове как конструкторов они будут создавать объекты одного класса — по определению;
- можно создать несколько объектов фабрично по одному и тому же прототипу, и это опять же сформирует класс объектов.

Вот пример фабричного создания объекта по прототипу, определённого в примерах выше:

```
const cb2 = Object.create(Cube.prototype, {
  x: {
    writable: true,
    configurable: true,
    value: 4
  }
});
console.log(cb2.x); // 4
console.log(cb2.volume()); // 64
```

Здесь мы указали прототип и собственное свойство (`x`) нашего объекта во втором аргументе метода `Object.create`. Это так называемый `propertiesObject` (объект с описанием свойств другого объекта). Если он определён, то его собственные перечислимые свойства (т.е. те, что определены в нём самом, а не перечислимые свойства в рамках его цепочки прототипов) определяют дескрипторы свойств, которым предстоит быть добавленным во

внось создаваемый объект, с соответствующими именами. Эти свойства соответствуют второму аргументу метода `Object.defineProperty`.

Однако, имя конструктора обычно используется в качестве имени класса, т.е. если прототип определяет поведение объекта, то конструктор определяет, так сказать, его фасад. В JavaScript существует оператор `instanceof`, правый операнд которого представляет собой имя конструктора.

Оператор `instanceof` проверяет, наследует ли объект `cb` свойство `Cube.prototype`. Он не проверяет, был ли объект `cb` инициализирован конструктором `Cube`.

```
console.log(cb2.constructor); // [Function: Cube]
console.log(cb2 instanceof Cube); // true
console.log(cb2 instanceof Object); // true
```

Тем не менее, правым операндом указывается именно конструктор.

Мы можем сначала создать объект без прототипа, потом назначить ему прототип, и результат будет тот же самый:

```
console.log(Reflect.getPrototypeOf(cb3)); // null
console.log(cb3.x); // 5
// console.log(cb3.volume()); not a Function
// только что метода не было, как и прототипа, а теперь будет
Reflect.setPrototypeOf(cb3, Reflect.getPrototypeOf(cb2));
console.log(cb3.volume()); // 125

console.log(cb3.constructor); // [Function: Cube]
console.log(cb3 instanceof Cube); // true
console.log(cb3 instanceof Object); // true
```

Классы ES2015

Стандарт ES2015 позволил перейти к созданию объектов с помощью синтаксиса классов. На самом деле «под капотом» всё равно происходит использование прототипов, но реализация наследования стала более удобной.

```

1  const Cube = function (x) { this.x = x; };
2  Cube.prototype.volume = function () { return this.x ** 3; };
3  const CubeClass = class {
4      constructor(x) {
5          this.x = x;
6      }
7      volume() {
8          return this.x ** 3;
9      }
10 };
11 const cb = new Cube(5);
12 const cbCl = new CubeClass(5);
13
14 console.log(cb.constructor); // Cube
15 console.log(cbCl.constructor); // CubeClass
16
17 const cbClProt = Reflect.getPrototypeOf(cbCl); // CubeClass {}
18 const cbProt = Reflect.getPrototypeOf(cb); // Cube { volume: [Function] }
19 console.log(cbClProt === CubeClass.prototype); // true
20 console.log(cbProt === Cube.prototype); // true

```

В этом примере показано, что классы просто собирают вместе то, что ранее писалось отдельно: свойства и методы. Методы по-прежнему находятся в прототипе, только при объявлении с помощью класса они перечислимы. Впрочем, это несложно переопределить:

```

const cbClProt = Reflect.getPrototypeOf(cbCl); // CubeClass {}
Reflect.defineProperty(cbClProt, 'volume', { enumerable: true });
console.log(Reflect.getPrototypeOf(cbCl)); // CubeClass { volume: [Function: volume] }

```

В примере выше определения класса в 1-2 строках и в 3-10 работают очень похоже, но не полностью эквивалентно, поскольку в 1-2 строках не утверждается перечислимость метода volume. Есть и ещё несколько отличий синтаксиса классов, в том числе:

-
- весь код в объявлении класса автоматически выполняется в строгом режиме и внутри классов нет никакой возможности выйти из строгого режима;
 - вызов конструктора класса без ключевого слова new завершается ошибкой;
 - попытка затереть имя класса внутри метода завершается ошибкой.

Если вызвать конструктор Cube без new, ничего страшного не произойдёт: просто будет вызвана функция Cube. Т.к. в ней нет return, этот вызов возвратит undefined.

В случае с синтаксисом классов попытка вызвать без new приведёт к ошибке: `TypeError: Class constructor CubeClass cannot be invoked without 'new'`.

Соответственно, чтобы учесть эти особенности, можно написать такую реализацию класса Cube на основе конструктора:

```

let Cube2 = (() => {
  'use strict';

  const Cube2 = function (x) {
    if (typeof new.target === 'undefined') {
      throw new Error('Use new!');
    }
    this.x = x;
  };
  Object.defineProperty(Cube2.prototype, 'volume', {
    value() {
      if (typeof new.target !== 'undefined') {
        throw new Error('Use new!');
      }
      return this.x ** 3;
    },
    enumerable: false,
    writable: true,
    configurable: true
  });
  return Cube2;
})();

```

Использование PFE позволяет собрать части класса воедино и защитить имя Cube2 от изменений внутри тела PFE (благодаря const), оставив возможность внешнему коду переписать переменную Cube2 (благодаря let).

```

1  /* eslint no-new: 0 */
2  /* eslint new-parens: 0 */
3  /* eslint prefer-const: 0 */
4  /* eslint no-func-assign: 0 */
5
6  let F = function F() {
7    F = 4;
8    console.log(F.name);
9  };
10 new F; // F
11
12 let G = class G {
13   constructor() {
14     // G = 4; // TypeError: Assignment to constant variable.
15     console.log(G.name);
16   }
17 };
18 new G; // G

```

В случае с функцией, если убрать F справа от function, результатом будет не F, а undefined. Это потому что инструкция let позволила переписать значение, а у числа нет свойства name. Свойство name у функции примет значение, указанное справа от function, если оно будет указано. Если после let стоит одно имя, а после function другое, то предпочтение будет отдано тому, которое стоит после let.

В строгом режиме попытка присвоить значение имени, указанном после function внутри функции вызывает ошибку, потому что внутри функции объявление function рассматривается как объявление константы.

```
'use strict';

let F = function FF() {
  // FF = 4; // TypeError: Assignment to constant variable.
  console.log(F.name);
};
new F; // FF
// new FF; FF is not defined
```

В классах, как говорилось выше, действует строгий режим. Поэтому попытка изменить имя, заданное словом class (если оно задано) вызовет ту же ошибку.

Но снаружи класса изменить его вполне возможно, если использовать let.

$$\frac{\text{let}}{\text{const}} X = \frac{\text{function}}{\text{class}} [Y]$$

Суммируя вышесказанное, слова function/class могут выступать в роли объявителей переменных самостоятельно, и в этом случае они создают имя, видимое как внутри функции/класса, так и снаружи, при этом class работает как const всегда, а function работает как const только в строгом режиме.

Если использовать let / const, то возможен случай указания имени после function/class и неуказания. В случае указания оно становится доступно только внутри функции/класса становясь значением X.name.
