1. **Методы объектов, this.**

Объекты обычно создаются, чтобы представлять сущности реального мира, будь то пользователи, заказы и так далее:

let user = {

  name: "Джон",

  age: 30

};

И так же, как и в реальном мире, пользователь может совершать действия: выбирать что-то из корзины покупок, авторизовываться, выходить из системы, оплачивать и т.п. Такие действия в JavaScript представлены свойствами-функциями объекта. Для начала добавим в объект user функцию приветствия:

let user = {

  name: "Джон",

  age: 30

};

user.sayHi = function() {

  alert("Привет!");

};

user.sayHi(); // Привет!

Здесь просто использовано Function Expression, чтобы создать функцию для приветствия, и присвоить её свойству user.sayHi объекта user. Затем она вызывается.

Функцию, которая является свойством объекта, называют *методом* этого объекта. Таким образом создан метод sayHi объекта user. Конечно, можно было бы заранее объявить функцию и использовать её в качестве метода, например так:

let user = {

  // ...

};

function sayHi() {

  alert("Привет!");

};

user.sayHi = sayHi;

user.sayHi(); // Привет!

Существует более короткий синтаксис для методов в литерале объекта:

user = {

  sayHi() {

    alert("Привет");

  }

};

Т.е., можно пропустить ключевое слово "function" и просто написать sayHi(). Нужно отметить, что эти две записи не полностью эквивалентны. Есть тонкие различия, связанные с наследованием объектов, но на данном этапе изучения это неважно. В большинстве случаев сокращённый синтаксис предпочтителен.

В JavaScript иногда необходимо сначала проверить, существует ли объект, а затем попытаться получить одно из его свойств, например, так:

const car = null;

const color = car && car.color;

Даже если car имеет значение null, у нас нет ошибок, а color присваивается значение null. Используя оператор &&, можно пройти несколько уровней вложенностей:

const car = {}

const colorName = car && car.color && car.color.name;

Оператор *опциональной последовательности* позволяет сделать код короче:

const car = {}

const color = car?.color;

const colorName = car?.color?.name;

Если car имеет значение null или undefined, результат будет undefined.

Как правило, методу объекта необходим доступ к информации, которая хранится в объекте, чтобы выполнить с ней какие-либо действия (в соответствии с назначением метода). Например, коду внутри user.sayHi() может понадобиться имя пользователя, которое хранится в объекте user. Для доступа к информации внутри объекта метод может использовать ключевое слово this. Значение this – это объект «перед точкой», который использовался для вызова метода. Например:

let user = {

  name: "Джон",

  age: 30,

  sayHi() {

    // this - это "текущий объект"

    alert(this.name);

  }

};

user.sayHi(); // Джон

Здесь во время выполнения кода user.sayHi() значением this будет являться user (ссылка на объект user). Технически также возможно получить доступ к объекту без ключевого слова this, ссылаясь на него через внешнюю переменную (в которой хранится ссылка на этот объект):

let user = {

  name: "Джон",

  age: 30,

  sayHi() {

    alert(user.name);

  }

};

Но такой код будет ненадёжным. Если скопировать ссылку на объект user в другую переменную, например, admin = user, и перезаписать переменную user чем-то другим, тогда будет осуществлён доступ к неправильному объекту при вызове метода из admin. Это показано ниже:

let user = {

  name: "Джон",

  age: 30,

  sayHi() {

    alert( user.name );

  }

};

let admin = user;

user = null;

admin.sayHi(); // Ошибка!

Если использовать this.name вместо user.name внутри alert, тогда этот код будет работать.

В JavaScript ключевое слово «this» ведёт себя иначе, чем в большинстве других языков программирования. Оно может использоваться в любой функции. В этом коде нет синтаксической ошибки:

function sayHi() {

  alert( this.name );

}

Значение this вычисляется во время выполнения кода и зависит от контекста. Например, здесь одна и та же функция назначена двум разным объектам и имеет различное значение «this» при вызовах:

let user = { name: "Джон" };

let admin = { name: "Админ" };

function sayHi() {

  alert( this.name );

}

user.f = sayHi;

admin.f = sayHi;

user.f(); // Джон  (this == user)

admin.f(); // Админ  (this == admin)

admin['f'](); // Админ

Правило простое: при вызове obj.f() значение this внутри f равно obj. Так что, в приведённом примере это user или admin.

Можно вызвать функцию вовсе без использования объекта:

function sayHi() {

  alert(this);

}

sayHi(); // undefined

В строгом режиме ("use strict") в таком коде значением this будет являться undefined. Если попытаться получить доступ к name, используя this.name – это вызовет ошибку.

В нестрогом режиме значением this в таком случае будет глобальный объект. Обычно подобный вызов является ошибкой программирования. Если внутри функции используется this, тогда ожидается, что она будет вызываться в контексте какого-либо объекта.

В других языках программирования this фиксировано – методы, определённые внутри объекта, всегда сохраняют в качестве значения this ссылку на свой объект (в котором был определён метод). В JavaScript this является «свободным», его значение вычисляется в момент вызова метода и не зависит от того, где этот метод был объявлен, а зависит от того, какой объект вызывает метод (какой объект стоит «перед точкой»).

Такая особенность вычисления this в момент исполнения имеет как свои плюсы, так и минусы. С одной стороны, функция может быть повторно использована в качестве метода у различных объектов (что повышает гибкость). С другой стороны, большая гибкость увеличивает вероятность ошибок.

Некоторые хитрые способы вызова метода приводят к потере значения this, например:

let user = {

  name: "Джон",

  hi() { alert(this.name); },

  bye() { alert("Пока"); }

};

user.hi();

 (user.name == "Джон" ? user.hi : user.bye)(); // Ошибка!

В последней строчке кода используется условный оператор ?, который определяет, какой будет вызван метод (user.hi или user.bye) в зависимости от выполнения условия. В данном случае будет выбран user.hi. Затем метод тут же вызывается с помощью скобок (). Но вызов не работает как положено: при вызове будет ошибка, потому что значением "this" внутри функции становится undefined (полагаем, что у нас строгий режим). Так работает (доступ к методу объекта через точку):

user.hi();

Так уже не работает (вызываемый метод вычисляется):

(user.name == "Джон" ? user.hi : user.bye)(); // Ошибка!

Чтобы понять, почему так происходит, разберёмся, как работает вызов методов (obj.method()). В выражении obj.method() сначала оператор точка '.' возвращает свойство объекта – его метод (obj.method). Затем скобки () вызывают этот метод (исполняется код метода). Если поместить эти операции в отдельные строки, то значение this, естественно, будет потеряно:

let user = {

  name: "Джон",

  hi() { alert(this.name); }

}

let hi = user.hi;

hi(); // Ошибка

Здесь hi = user.hi сохраняет функцию в переменной, и далее в последней строке она вызывается полностью сама по себе, без объекта, так что нет this.

Для работы вызовов типа user.hi(), JavaScript использует трюк – точка '.' возвращает не саму функцию, а специальное значение «ссылочного типа», называемого [Reference Type](https://tc39.github.io/ecma262/" \l "sec-reference-specification-type). Этот ссылочный тип является внутренним типом. Нельзя явно использовать его, но он используется внутри языка. Значение ссылочного типа – это «триплет»: комбинация из трех значений (base, name, strict), где:

* base – это объект.
* name – это имя свойства объекта.
* strict – это режим исполнения. Является true, если действует строгий режим (use strict).

Результатом доступа к свойству user.hi является не функция, а значение ссылочного типа. Для user.hi в строгом режиме оно будет таким:

// значение ссылочного типа (Reference Type)

(user, "hi", true)

Когда скобки () применяются к значению ссылочного типа (происходит вызов), то они получают полную информацию об объекте и его методе, и могут поставить правильный this (=user в данном случае, по base).

Ссылочный тип – исключительно внутренний, промежуточный, используемый, чтобы передать информацию от точки .до вызывающих скобок (). При любой другой операции, например, присваивании hi = user.hi, ссылочный тип заменяется на собственно значение user.hi (функцию), и дальше работа уже идёт только с ней. Поэтому дальнейший вызов происходит уже без this. Таким образом, значение this передаётся правильно, только если функция вызывается напрямую с использованием синтаксиса точки obj.method() или квадратных скобок obj['method']() (они делают то же самое).

Стрелочные функции особенные: у них нет своего «собственного» this. Если использовать this внутри стрелочной функции, то его значение берётся из внешней обычной функции. Например, здесь arrow() использует значение this из внешнего метода user.sayHi():

let user = {

  firstName: "Вася",

  sayHi() {

    let arrow = () => alert(this.firstName);

    arrow();

  }

};

user.sayHi(); // Вася

Это является особенностью стрелочных функций. Они полезны, когда нет необходимости иметь отдельное значение this, а надо брать его из внешнего контекста.

Обычный синтаксис {...} позволяет создать только один объект. Но часто нужно создать множество однотипных объектов, таких как пользователи, элементы меню и т.д. Это можно сделать при помощи функции-конструктора и оператора "new".

[**Функция-конструктор**](https://learn.javascript.ru/constructor-new#funktsiya-konstruktor)

Функции-конструкторы являются обычными функциями. Но есть два соглашения:

1. Имя функции-конструктора должно начинаться с большой буквы.
2. Функция-конструктор должна вызываться при помощи оператора "new".

Например:

function User(name) {

  this.name = name;

  this.isAdmin = false;

}

let user = new User("Вася");

alert(user.name); // Вася

alert(user.isAdmin); // false

Когда функция вызывается как new User(...), происходит следующее:

1. Создаётся новый пустой объект, и он присваивается this.
2. Выполняется код функции. Обычно он модифицирует this, добавляет туда новые свойства.
3. Возвращается значение this.

Другими словами, вызов new User(...) делает примерно вот что:

function User(name) {

  // this = {};

  this.name = name;

  this.isAdmin = false;

  // return this;

}

То есть, результат вызова new User("Вася") – это тот же объект, что и:

let user = {

  name: "Вася",

  isAdmin: false

};

Теперь, когда необходимо будет создать других пользователей, можно использовать new User("Маша"), new User("Даша") и т.д. Данная конструкция гораздо удобнее и читабельнее, чем каждый раз создавать литерал объекта. Это и является основной целью конструкторов – удобное повторное создание однотипных объектов.

Ещё раз заметим: технически, любая функция может быть использована как конструктор. То есть, каждая функция может быть вызвана при помощи оператора new и выполнит алгоритм, указанный выше в примере. Заглавная буква в названии функции является всеобщим соглашением по именованию, она как бы подсказывает разработчику, что данная функция является функцией-конструктором и её нужно вызывать через new.

**new function() { … }**

Если коде большое количество строк, создающих один сложный объект, можно обернуть их в функцию-конструктор следующим образом:

let user = new function() {

  this.name = "Вася";

  this.isAdmin = false;

  // ...

};

Такой конструктор не может быть вызван дважды, так как он нигде не сохраняется, просто создаётся и тут же вызывается. Таким образом, такой метод создания позволяет инкапсулировать код, который создаёт отдельный объект, но без возможности его повторного использования. Данный метод используется очень редко.

Используя специальное свойство new.target внутри функции, можно проверить, вызвана ли функция при помощи оператора new или без него.

Обычно конструкторы ничего не возвращают. Их задача – записать все необходимое в this, который в итоге станет результатом.

Но если return всё же есть, то применяется простое правило:

* при вызове return с объектом, будет возвращён объект, а не this;
* при вызове return с примитивным значением, примитивное значение будет отброшено.

Другими словами, return с объектом возвращает объект, в любом другом случае конструктор вернёт this. В примере ниже return возвращает объект вместо this:

function BigUser() {

  this.name = "Вася";

  // возвращает объект

  return { name: "Godzilla" };

}

alert( new BigUser().name );

Пример с пустым return (можно поставить примитив после return, не важно):

function SmallUser() {

  this.name = "Вася";

  return; // возвращает this

  // ...

}

alert( new SmallUser().name );  // Вася

Можно не ставить скобки после new, если вызов конструктора идёт без аргументов:

let user = new User;

// то же, что и

let user = new User();

Пропуск скобок считается плохой практикой, но синтаксис языка такое позволяет.

[**Создание методов в конструкторе**](https://learn.javascript.ru/constructor-new#sozdanie-metodov-v-konstruktore)

Использование конструкторов для создания объектов даёт большую гибкость. Можно передавать конструктору параметры, определяющие, как создавать объект, и что в него записывать. В this можно добавлять не только свойства, но и методы.

Например, в примере ниже, new User(name) создаёт объект с данным именем name и методом sayHi:

function User(name) {

  this.name = name;

  this.sayHi = function() {

    alert( "Меня зовут: " + this.name );

  };

}

let vasya = new User("Вася");

vasya.sayHi(); // Меня зовут: Вася

/\*

vasya = {

   name: "Вася",

   sayHi: function() { ... }

}

\*/

1. **Флаги и дескрипторы свойств.**

[**Флаги свойств**](https://learn.javascript.ru/property-descriptors#flagi-svoystv)

Помимо значения value, свойства объекта имеют три специальных атрибута (так называемые «флаги»):

* writable – если true, свойство можно изменить, иначе оно только для чтения.
* enumerable – если true, свойство перечисляется в циклах, в противном случае циклы его игнорируют.
* configurable – если true, свойство можно удалить, а эти атрибуты можно изменять, иначе этого делать нельзя.

Эти атрибуты обычно скрыты. Когда создается свойство «обычным способом», все эти атрибуты имеют значение true. Но можно изменить их в любое время.

Метод [Object.getOwnPropertyDescriptor](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptor) позволяет получить полную информацию о свойстве. Его синтаксис:

let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);

* obj – объект, из которого получаем информацию.
* propertyName – имя свойства.

Возвращаемое значение – это объект, так называемый «дескриптор свойства»: он содержит значение свойства и все его флаги. Например:

let user = {

  name: "John"

};

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );

/\* дескриптор свойства:

{

  "value": "John",

  "writable": true,

  "enumerable": true,

  "configurable": true

}

\*/

Чтобы изменить флаги, можно использовать метод [Object.defineProperty](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty). Его синтаксис:

Object.defineProperty(obj, propertyName, descriptor)

* obj, propertyName – объект и его свойство, для которого нужно применить дескриптор.
* descriptor – применяемый дескриптор.

Если свойство существует, defineProperty обновит его флаги. В противном случае метод создает новое свойство с указанным значением и флагами; если какой-либо флаг не указан явно, ему присваивается значение false. Например, здесь создаётся свойство name, все флаги которого имеют значение false:

let user = {};

Object.defineProperty(user, "name", {

  value: "John"

});

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );

/\*

{

  "value": "John",

  "writable": false,

  "enumerable": false,

  "configurable": false

}

 \*/

Сравните этот способ с user.name, который создан выше «обычным способом»: в этот раз все флаги имеют значение false. Если это не то, что нужно, надо присвоить им значения true в параметре descriptor.

Рассмотрим на примерах, что даёт использование флагов.

[**Только для чтения**](https://learn.javascript.ru/property-descriptors#tolko-dlya-chteniya)

Сделаем свойство user.name доступным только для чтения. Для этого изменим флаг writable:

let user = {

  name: "John"

};

Object.defineProperty(user, "name", {

  writable: false

});

user.name = "Pete"; // Ошибка

Теперь никто не сможет изменить имя пользователя, если только не обновит соответствующий флаг новым вызовом defineProperty.

Ошибки появляются только в строгом режиме, в нестрогом режиме, без use strict, не видно никаких ошибок при записи в свойства «только для чтения» и т.п. Но эти операции всё равно не будут выполнены успешно. Действия, нарушающие ограничения флагов, в нестрогом режиме просто молча игнорируются.

Вот тот же пример, но свойство создано «с нуля»:

let user = { };

Object.defineProperty(user, "name", {

  value: "John",

  enumerable: true,

  configurable: true

});

alert(user.name); // John

user.name = "Pete"; // Ошибка

**[Неперечисляемое свойство](https://learn.javascript.ru/property-descriptors" \l "neperechislimoe-svoystvo)**

Добавим собственный метод toString к объекту user. Встроенный метод toString в объектах – неперечисляемый, его не видно в цикле for..in. Но если написать свой собственный метод toString, цикл for..in будет выводить его по умолчанию:

let user = {

  name: "John",

  toString() {

    return this.name;

  }

};

for (let key in user) alert(key); // name, toString

Если в этом нет необходимости, можно установить для свойства enumerable: false. Тогда оно перестанет появляться в цикле for..in, аналогично встроенному toString:

let user = {

  name: "John",

  toString() {

    return this.name;

  }

};

Object.defineProperty(user, "toString", {

  enumerable: false

});

for (let key in user) alert(key); // name

Неперечисляемые свойства также не возвращаются Object.keys:

alert(Object.keys(user)); // name

[**Неконфигурируемое свойство**](https://learn.javascript.ru/property-descriptors#nekonfiguriruemoe-svoystvo)

Флаг неконфигурируемого свойства (configurable: false) иногда предустановлен для некоторых встроенных объектов и свойств. Неконфигурируемое свойство не может быть удалено или изменено с помощью defineProperty. Например, свойство Math.PI – только для чтения, неперечисляемое и неконфигурируемое:

let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');

alert( JSON.stringify(descriptor, null, 2 ) );

/\*

{

  "value": 3.141592653589793,

  "writable": false,

  "enumerable": false,

  "configurable": false

}

\*/

То есть программист не сможет изменить значение Math.PI или перезаписать его.

Math.PI = 3; // Ошибка

Если свойство определено как неконфигурируемое, то нельзя поменять его обратно, потому что defineProperty не работает с неконфигурируемыми свойствами. В коде ниже свойство name определено как константа:

let user = { };

Object.defineProperty(user, "name", {

  value: "John",

  writable: false,

  configurable: false

});

Object.defineProperty(user, "name", {writable: true}); // Ошибка

В нестрогом режиме мы не увидим никаких ошибок при записи в свойства «только для чтения» и т.п. Эти операции всё равно не будут выполнены успешно. Действия, нарушающие ограничения флагов, в нестрогом режиме просто молча игнорируются.

[**Метод Object.defineProperties**](https://learn.javascript.ru/property-descriptors#metod-object-defineproperties)

Существует метод [Object.defineProperties(obj, descriptors)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperties), который позволяет определять множество свойств сразу. Его синтаксис:

Object.defineProperties(obj, {

  prop1: descriptor1,

  prop2: descriptor2

  // ...

});

Например:

Object.defineProperties(user, {

  name: { value: "John", writable: false },

  surname: { value: "Smith", writable: false },

  // ...

});

Таким образом, можно определить множество свойств одной операцией.

**Метод** [**Object.getOwnPropertyDescriptors**](https://learn.javascript.ru/property-descriptors#object-getownpropertydescriptors)

Чтобы получить все дескрипторы свойств сразу, можно воспользоваться методом [Object.getOwnPropertyDescriptors(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptors). Вместе с Object.defineProperties этот метод можно использовать для клонирования объекта вместе с его флагами:

let clone = Object.defineProperties({}, Object.getOwnPropertyDescriptors(obj));

Обычно при клонировании объекта используется присваивание, чтобы скопировать его свойства:

for (let key in user) {

  clone[key] = user[key]

}

Но это не копирует флаги. Поэтому если нужен клон с флагами,  предпочтительнее использовать Object.defineProperties. Другое отличие в том, что for..in игнорирует символьные свойства, а Object.getOwnPropertyDescriptors возвращает дескрипторы всех свойств, включая свойства-символы.

[**Глобальное запечатывание объекта**](https://learn.javascript.ru/property-descriptors#globalnoe-zapechatyvanie-obekta)

Дескрипторы свойств работают на уровне конкретных свойств. Но еще есть методы, которые ограничивают доступ ко всему объекту:

* [Object.preventExtensions(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/preventExtensions) – запрещает добавлять новые свойства в объект.
* [Object.seal(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/seal) – запрещает добавлять/удалять свойства. Устанавливает configurable: false для всех существующих свойств.
* [Object.freeze(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze) – запрещает добавлять/удалять/изменять свойства. Устанавливает configurable: false, writable: false для всех существующих свойств.

А также есть методы для их проверки:

* [Object.isExtensible(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/isExtensible) – возвращает false, если добавление свойств запрещено, иначе true.
* [Object.isSealed(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/isSealed) – возвращает true, если добавление/удаление свойств запрещено и для всех существующих свойств установлено configurable: false.
* [Object.isFrozen(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/isFrozen) – возвращает true, если добавление/удаление/изменение свойств запрещено, и для всех текущих свойств установлено configurable: false, writable: false.

На практике эти методы используются редко.

1. **Геттеры и сеттеры.**

Есть два типа свойств объекта. Первый тип это свойства-данные (data properties). Все свойства, которые использовались до текущего момента были свойствами-данными. Второй тип свойств это свойства-аксессоры (accessor properties). По своей сути это функции, которые используются для присвоения и получения значения, но во внешнем коде они выглядят как обычные свойства объекта.

[**Геттеры и сеттеры**](https://learn.javascript.ru/property-accessors#gettery-i-settery)

Свойства-аксессоры представлены методами: «геттер» – для чтения и «сеттер» – для записи. При литеральном объявлении объекта они обозначаются get и set:

let obj = {

  get propName() {

    // геттер, срабатывает при чтении obj.propName

  },

  set propName(value) {

    // сеттер, срабатывает при записи obj.propName = value

  }

};

Геттер срабатывает, когда obj.propName читается, сеттер – когда значение назначается. Например, есть объект user со свойствами name и surname:

let user = {

  name: "John",

  surname: "Smith"

};

Добавим свойство объекта fullName для полного имени – "John Smith". Реализуем его при помощи аксессора:

let user = {

  name: "John",

  surname: "Smith",

  get fullName() {

    return `${this.name} ${this.surname}`;

  }

};

alert(user.fullName); // John Smith

Снаружи свойство-аксессор выглядит как обычное свойство. В этом и заключается смысл свойств-аксессоров. user.fullName  не вызывается  как функция, а читается как обычное свойство: геттер сам вернет нужное значение.

На данный момент в примере fullName имеет только геттер. Если попытаться присвоить значение свойству user.fullName, то это вызовет ошибку:

let user = {

  get fullName() {

    return `...`;

  }

};

user.fullName = "Тест"; // Ошибка

Давайте исправим это, добавив сеттер для user.fullName:

let user = {

  name: "John",

  surname: "Smith",

  get fullName() {

    return `${this.name} ${this.surname}`;

  },

  set fullName(value) {

    [this.name, this.surname] = value.split(" ");

  }

};

user.fullName = "Alice Cooper";

alert(user.name); // Alice

alert(user.surname); // Cooper

В итоге получим «виртуальное» свойство fullName. Его можно прочитать и изменить, но по факту его не существует.

При попытке удалить свойство-аксессор оператором delete будет ошибка.

Дескрипторы свойств-аксессоров отличаются от «обычных» свойств-данных. Свойства-аксессоры не имеют value и writable, но взамен предлагают функции get и set.

То есть, дескриптор аксессора может иметь:

* get – функция без аргументов, которая сработает при чтении свойства,
* set – функция, принимающая один аргумент, вызываемая при присвоении свойства,
* enumerable – то же самое, что и для свойств-данных,
* configurable – то же самое, что и для свойств-данных.

Например, для создания аксессора fullName при помощи defineProperty можно передать дескриптор с использованием get и set:

let user = {

  name: "John",

  surname: "Smith"

};

Object.defineProperty(user, 'fullName', {

  get() {

    return `${this.name} ${this.surname}`;

  },

  set(value) {

    [this.name, this.surname] = value.split(" ");

  }

});

alert(user.fullName); // John Smith

for(let key in user) alert(key); // name, surname

Ещё раз заметим, что свойство объекта может быть только свойством-аксессором (с методами get/set) или свойством-данных (со значением value). При попытке указать и get и value в одном дескрипторе будет ошибка:

// Error: Invalid property descriptor.

Object.defineProperty({}, 'prop', {

  get() {

    return 1

  },

  value: 2

});

Геттеры/сеттеры можно использовать как обёртки над «реальными» значениями свойств, чтобы получить больше контроля над операциями с ними. Например, если надо запретить устанавливать короткое имя для user, можно использовать сеттер name для проверки, а само значение хранить в отдельном свойстве \_name:

let user = {

  get name() {

    return this.\_name;

  },

  set name(value) {

    if (value.length < 4) {

      alert("Имя слишком короткое, должно быть более 4 символов");

      return;

    }

    this.\_name = value;

  }

};

user.name = "Pete";

alert(user.name); // Pete

user.name = ""; // Имя слишком короткое...

Таким образом, само имя хранится в \_name, доступ к которому производится через геттер и сеттер. Технически, внешний код всё ещё может получить доступ к имени напрямую с помощью user.\_name, но существует широко известное соглашение о том, что свойства, которые начинаются с символа "\_", являются внутренними, и к ним не следует обращаться извне пределов объекта.

Аксессоры позволяют в любой момент взять «обычное» свойство и изменить его поведение, поменяв на геттер и сеттер. Например, представим, что реализован объект user, с использованием свойств-данных имя name и возраст age:

function User(name, age) {

  this.name = name;

  this.age = age;

}

let john = new User("John", 25);

alert( john.age ); // 25

Но со временем взамен возраста age можно хранить дату рождения birthday, потому что так более точно и удобно:

function User(name, birthday) {

  this.name = name;

  this.birthday = birthday;

}

let john = new User("John", new Date(1992, 6, 1));

Чтобы не менять весь старый код, который использует свойство age можно добавить геттер для age:

function User(name, birthday) {

  this.name = name;

  this.birthday = birthday;

  Object.defineProperty(this, "age", {

    get() {

      let todayYear = new Date().getFullYear();

      return todayYear - this.birthday.getFullYear();

    }

  });

}

let john = new User("John", new Date(1992, 6, 1));

alert( john.birthday );

alert( john.age );

Теперь и старый код работает, и появилось полезное дополнительное свойство.