1. **`**[**Прототипное наследование**](https://learn.javascript.ru/prototype-inheritance)

В программировании часто возникает необходимость что-то расширить. Например, есть объект user со своими свойствами и методами, надо создать объекты admin и guest как его слегка изменённые варианты. Хотелось бы повторно использовать то, что есть у объекта user, не копировать/переопределять его методы, а просто создать новый объект на его основе.

*Прототипное наследование* — это возможность языка, которая помогает в этом.

**Свойство** [**[[Prototype]]**](https://learn.javascript.ru/prototype-inheritance#prototype)

В JavaScript объекты имеют специальное скрытое свойство [[Prototype]] (так оно названо в спецификации), которое либо равно null, либо ссылается на другой объект. Этот объект называется «прототип».

Если при чтении свойство из object отсутствует, JavaScript автоматически берет его из прототипа. В программировании такой механизм называется *прототипным наследованием*. Многие возможности языка и техники программирования основываются на нем.

Свойство [[Prototype]] является внутренним и скрытым, но есть много способов задать его. Одним из них является использование \_\_proto\_\_, например так:

let animal = {

  eats: true

};

let rabbit = {

  jumps: true

};

rabbit.\_\_proto\_\_ = animal;

 Свойство \_\_proto\_\_ – не то же самое, что [[Prototype]]. Это геттер/сеттер для него. Он существует по историческим причинам, в современном языке его заменяют функции Object.getPrototypeOf/Object.setPrototypeOf, которые также получают/устанавливают прототип. По спецификации \_\_proto\_\_ должен поддерживаться только браузерами, но по факту все среды, включая серверную, поддерживают его.

В примере ниже осуществляется поиск свойства в rabbit, а оно отсутствует, и JavaScript автоматически берет его из animal:

let animal = {

  eats: true

};

let rabbit = {

  jumps: true

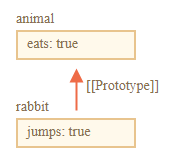
};

rabbit.\_\_proto\_\_ = animal; // (\*)

alert( rabbit.eats ); // true (\*\*)

alert( rabbit.jumps ); // true

Здесь строка (\*) устанавливает animal как прототип для rabbit. Затем, когда alert пытается прочитать свойство rabbit.eats (\*\*), его нет в rabbit, поэтому JavaScript следует по ссылке [[Prototype]] и находит её в animal (смотрите снизу вверх):



Здесь можно сказать, что animal является прототипом rabbit или rabbit прототипно наследует от animal. Так что если у animal много полезных свойств и методов, то они автоматически становятся доступными у rabbit. Такие свойства называются *унаследованными*. Например, есть метод в animal, он может быть вызван на rabbit:

let animal = {

  eats: true,

  walk() {

    alert("Animal walk");

  }

};

let rabbit = {

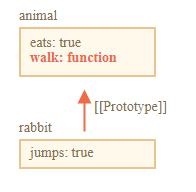
  jumps: true,

  \_\_proto\_\_: animal

};

rabbit.walk(); // Animal walk

Метод автоматически берётся из прототипа:



Цепочка прототипов может быть длиннее:

let animal = {

  eats: true,

  walk() {

    alert("Animal walk");

  }

};

let rabbit = {

  jumps: true,

  \_\_proto\_\_: animal

};

let longEar = {

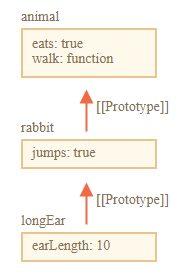
  earLength: 10,

  \_\_proto\_\_: rabbit

};

longEar.walk(); // Animal walk

alert(longEar.jumps); // true (для rabbit)



Есть только два ограничения:

1. Ссылки не могут идти по кругу. JavaScript выдаст ошибку, если попытаться назначить \_\_proto\_\_ по кругу.
2. Значение \_\_proto\_\_ может быть объектом или null. Другие типы игнорируются.

Это вполне очевидно, но все же: может быть только один [[Prototype]]. Объект не может наследовать от двух других.

Прототип используется только для чтения свойств. Операции записи/удаления работают напрямую с объектом. В приведённом ниже примере присваивается rabbit собственный метод walk:

let animal = {

  eats: true,

  walk() {

    /\* ... \*/

  }

};

let rabbit = {

  \_\_proto\_\_: animal

};

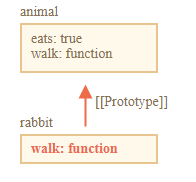
rabbit.walk = function() {

  alert("Rabbit! Bounce-bounce!");

};

rabbit.walk(); // Rabbit! Bounce-bounce!

Теперь вызов rabbit.walk() находит метод непосредственно в объекте и выполняет его, не используя прототип:



Свойства-акссессоры – исключение, так как запись в него обрабатывается функцией-сеттером. То есть, это, фактически, вызов функции. По этой причине admin.fullName работает корректно в приведённом ниже коде:

let user = {

  name: "John",

  surname: "Smith",

  set fullName(value) {

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

  },

  get fullName() {

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

  }

};

let admin = {

  \_\_proto\_\_: user,

  isAdmin: true

};

alert(admin.fullName); // John Smith (\*)

// срабатывает сеттер!

admin.fullName = "Alice Cooper"; // (\*\*)

Здесь в строке (\*) свойство admin.fullName имеет геттер в прототипе user, поэтому вызывается он. В строке (\*\*) свойство также имеет сеттер в прототипе, который и будет вызван.

[**Значение «this»**](https://learn.javascript.ru/prototype-inheritance#znachenie-this)

Прототипы никак не влияют на this. Неважно, где находится метод: в объекте или его прототипе. При вызове метода this – всегда объект перед точкой. Таким образом, вызов сеттера admin.fullName в качестве this использует admin, а не user.

Это на самом деле очень важная деталь, потому что может быть большой объект со множеством методов, от которого можно наследовать. Затем наследущие объекты могут вызывать его методы, но они будут изменять состояние этих объектов, а не большого. Например, здесь animal представляет собой «хранилище методов», и rabbit использует его. Вызов rabbit.sleep() устанавливает this.isSleeping для объекта rabbit:

let animal = {

  walk() {

    if (!this.isSleeping) {

      alert(`I walk`);

    }

  },

  sleep() {

    this.isSleeping = true;

  }

};

let rabbit = {

  name: "White Rabbit",

  \_\_proto\_\_: animal

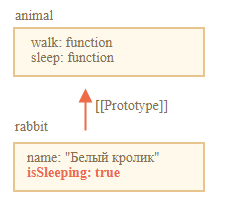
};

rabbit.sleep();

alert(rabbit.isSleeping); // true

alert(animal.isSleeping); // undefined (нет такого свойства в прототипе)

Картинка с результатом:



Если бы были другие объекты, такие как bird, snake и т.д., унаследованные от animal, они также получили бы доступ к методам animal. Но this при вызове каждого метода будет соответствовать объекту, на котором происходит вызов (перед точкой), а не animal. Поэтому, когда записываются данные в this, они сохраняются в этих объектах. В результате методы являются общими, а состояние объекта — нет.

[**Цикл for…in**](https://learn.javascript.ru/prototype-inheritance#tsikl-for-in)

Цикл for..in проходит не только по собственным, но и по унаследованным свойствам объекта. Например:

let animal = {

  eats: true

};

let rabbit = {

  jumps: true,

  \_\_proto\_\_: animal

};

alert(Object.keys(rabbit)); // jumps

for(let prop in rabbit) alert(prop); // jumps, then eats

Если унаследованные свойства не нужны, то можно отфильтровать их при помощи встроенного метода [obj.hasOwnProperty(key)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty): он возвращает true, если у obj есть собственное, не унаследованное, свойство с именем key. Пример такой фильтрации:

let animal = {

  eats: true

};

let rabbit = {

  jumps: true,

  \_\_proto\_\_: animal

};

for(let prop in rabbit) {

  let isOwn = rabbit.hasOwnProperty(prop);

  if (isOwn) {

    alert(`Our: ${prop}`); // Our: jumps

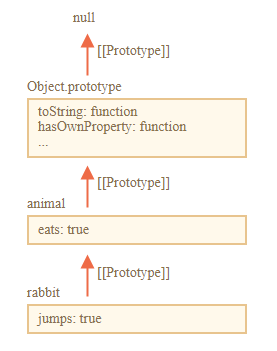
  } else {

    alert(`Inherited: ${prop}`); // Inherited: eats

  }

}

В этом примере цепочка наследования выглядит так: rabbit наследует от animal, который наследует от Object.prototype (так как animal – литеральный объект {...}, это по умолчанию), а затем null на самом верху:



Стоит отметить следующее: метод rabbit.hasOwnProperty явно не определен. Если посмотреть на цепочку прототипов, то видно, что он берётся из Object.prototype.hasOwnProperty. То есть, он унаследован, но не появляется в цикле for..in, в отличие от eats и jumps. Дело в том, что это свойство не перечислимо. То есть, у него внутренний флаг enumerable стоит false, как и у других свойств Object.prototype. Поэтому оно и не появляется в цикле.

Почти все методы, получающие ключи/значения, такие как Object.keys, Object.values и другие – игнорируют унаследованные свойства. Они учитывают только свойства самого объекта, не его прототипа.

1. [**F.prototype**](https://learn.javascript.ru/function-prototype)

Как известно, новые объекты могут быть созданы с помощью функции-конструктора, new F(). Если в F.prototype содержится объект, оператор new устанавливает его в качестве [[Prototype]] для нового объекта.

JavaScript использовал прототипное наследование с момента своего появления. Это одна из основных особенностей языка.

Но раньше, прямого доступа к прототипу объекта не было. Надёжно работало только свойство "prototype" функции-конструктора. Поэтому оно используется во многих скриптах. Обратите внимание, что F.prototype означает обычное свойство с именем "prototype" для F. Это ещё не «прототип объекта», а обычное свойство F с таким именем. Приведём пример:

let animal = {

  eats: true

};

function Rabbit(name) {

  this.name = name;

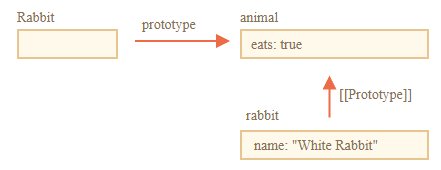
}

Rabbit.prototype = animal;

let rabbit = new Rabbit("White Rabbit"); //  rabbit.\_\_proto\_\_ == animal

alert( rabbit.eats ); // true

Установка Rabbit.prototype = animal буквально говорит интерпретатору следующее: "При создании объекта через new Rabbit() запиши ему animal в [[Prototype]]". Результат будет выглядеть так:



На изображении: "prototype" – горизонтальная стрелка, обозначающая обычное свойство для "F", а [[Prototype]]– вертикальная, обозначающая наследование rabbit от animal.

F.prototype используется только при вызове new F() и присваивается в качестве свойства [[Prototype]]нового объекта. После этого F.prototype и новый объект ничего не связывает. После создания F.prototype может измениться, и новые объекты, созданные с помощью new F(), будут иметь другой объект в качестве [[Prototype]], но уже существующие объекты сохранят старый.

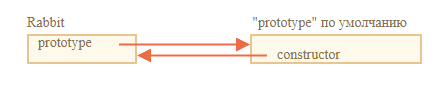
У каждой функции по умолчанию уже есть свойство "prototype". По умолчанию "prototype" – объект с единственным свойством constructor, которое ссылается на функцию-конструктор. Вот такой:

function Rabbit() {}

/\* прототип по умолчанию

Rabbit.prototype = { constructor: Rabbit };

\*/



Проверим это:

function Rabbit() {}

// по умолчанию:

// Rabbit.prototype = { constructor: Rabbit }

alert( Rabbit.prototype.constructor == Rabbit ); // true

Соответственно, если ничего не меняется, то свойство constructor будет доступно всем кроликам через [[Prototype]]:

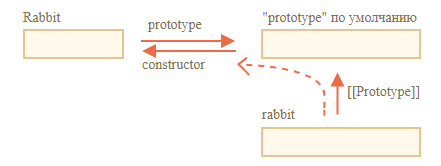
function Rabbit() {}

// по умолчанию:

// Rabbit.prototype = { constructor: Rabbit }

let rabbit = new Rabbit(); // наследует от {constructor: Rabbit}

alert(rabbit.constructor == Rabbit); // true (свойство получено из прототипа)



Можно использовать свойство constructor существующего объекта для создания нового. Пример:

function Rabbit(name) {

  this.name = name;

  alert(name);

}

let rabbit = new Rabbit("White Rabbit");

let rabbit2 = new rabbit.constructor("Black Rabbit");

Это удобно, когда есть объект, но неизвестно какой конструктор использовался для его создания (например, он был взят из сторонней библиотеки), а необходимо создать ещё один такой объект.

Самое важное о свойстве "constructor" это то, что JavaScript сам по себе не гарантирует правильное значение свойства "constructor". Оно является свойством по умолчанию в "prototype" у функций, но что будет с ним позже – зависит только от разработчика. В частности, если заменить прототип по умолчанию на другой объект, свойства "constructor" в нём не будет. Например:

function Rabbit() {}

Rabbit.prototype = {

  jumps: true

};

let rabbit = new Rabbit();

alert(rabbit.constructor === Rabbit); // false

Таким образом, чтобы сохранить верное свойство "constructor", надо добавлять/удалять/изменять свойства у прототипа по умолчанию вместо того, чтобы перезаписывать его целиком. В примере ниже Rabbit.prototype не перезаписывается полностью, а добавляется к нему свойство. Прототип по умолчанию сохраняется, и сохраняется доступ к Rabbit.prototype.constructor.

function Rabbit() {}

Rabbit.prototype.jumps = true

Или можно заново создать свойство constructor:

Rabbit.prototype = {

  jumps: true,

  constructor: Rabbit

};

1. [**Встроенные прототипы**](https://learn.javascript.ru/native-prototypes)

Свойство "prototype" широко используется в внутри самого языка JavaScript. Все встроенные функции-конструкторы используют его.

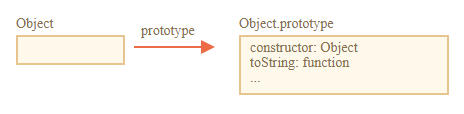
[**Object.prototype**](https://learn.javascript.ru/native-prototypes#object-prototype)

Выведем пустой объект:

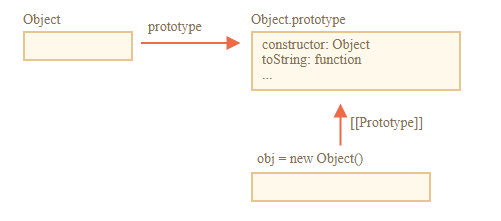
let obj = {};

alert( obj ); // "[object Object]" ?

В рассматриваемом примере нету кода, который генерирует строку "[object Object]". Понятно, что это встроенный метод toString, но явно не видно, где он объявлен,  ведь obj  пуст. Дело в том, что краткая нотация obj = {} это то же самое, что и obj = new Object(), где Object – встроенная функция-конструктор для объектов с собственным свойством prototype, который ссылается на огромный объект с методом toString и другими. Вот что происходит:



Когда вызывается new Object() (или создаётся объект с помощью литерала {...}), свойство [[Prototype]] этого объекта устанавливается на Object.prototype по правилам, которые рассматривались в предыдущем вопросе:



Таким образом, когда вызывается obj.toString(), метод берётся из Object.prototype. Можно проверить это так:

let obj = {};

alert(obj.\_\_proto\_\_ === Object.prototype); // true

// obj.toString === obj.\_\_proto\_\_.toString == Object.prototype.toString

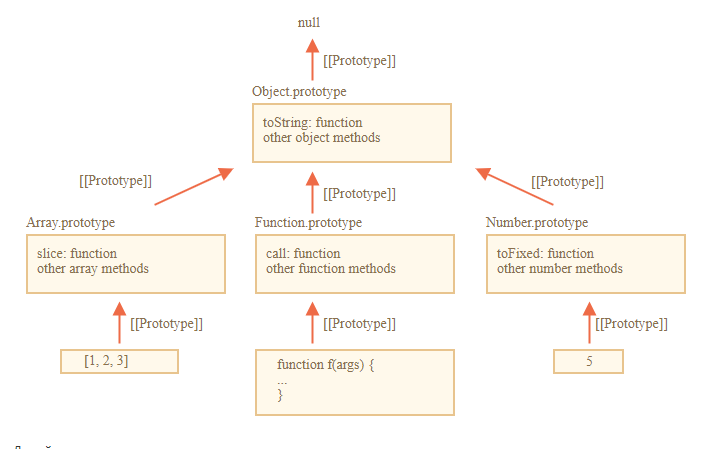
Обратим внимание, что выше Object.prototype по цепочке прототипов больше нет [[Prototype]]:

alert(Object.prototype.\_\_proto\_\_); // null

Другие встроенные объекты, такие как Array, Date, Function и другие, также хранят свои методы в прототипах. Например, при создании массива [1, 2, 3] внутренне используется конструктор массива Array. Поэтому прототипом массива становится Array.prototype, предоставляя ему свои методы. Это позволяет эффективно использовать память.

Согласно спецификации, наверху иерархии встроенных прототипов находится Object.prototype. Поэтому иногда говорят, что «всё наследует от объектов».

Вот более полная картина (для 3 встроенных объектов):



Проверим прототипы:

let arr = [1, 2, 3];

// наследует от Array.prototype?

alert( arr.\_\_proto\_\_ === Array.prototype ); // true

// затем от Object.prototype?

alert( arr.\_\_proto\_\_.\_\_proto\_\_ === Object.prototype ); // true

// и null на вершине иерархии

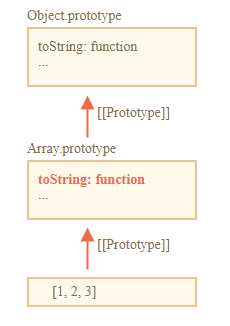
alert( arr.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_ ); // null

Некоторые методы в прототипах могут пересекаться, например, у Array.prototype есть свой метод toString, который выводит элементы массива через запятую:

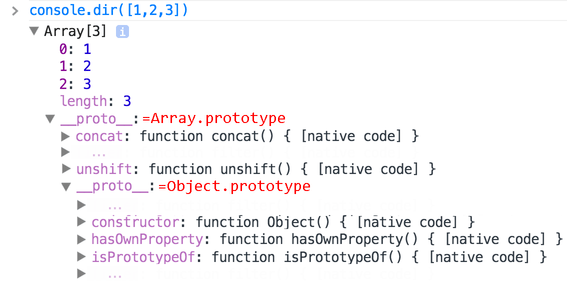
let arr = [1, 2, 3]

alert(arr); // 1,2,3 <-- результат Array.prototype.toString

Как известно, у Object.prototype есть свой метод toString, но так как Array.prototype ближе в цепочке прототипов, то берётся именно вариант для массивов:



В браузерных инструментах, таких как консоль разработчика, можно посмотреть цепочку наследования (возможно, потребуется использовать console.dir для встроенных объектов):



Другие встроенные объекты устроены аналогично. Даже функции – объекты встроенного конструктора Function, и все их методы (call/apply и другие) берутся из Function.prototype. Также у функций есть свой метод toString.

function f() {}

alert(f.\_\_proto\_\_ == Function.prototype); // true

alert(f.\_\_proto\_\_.\_\_proto\_\_ == Object.prototype); // true, наследует от Object

[**Примитивы**](https://learn.javascript.ru/native-prototypes#primitivy)

Самое сложное происходит со строкам, числами и булевым типом. Как известно, они не объекты. Но если попытаться получить доступ к их свойствам, тогда будет создан временный объект-обёртка с использованием встроенных конструкторов String, Number, Boolean, который предоставит методы и после чего исчезнет. Эти объекты создаются невидимо для нас, и большая часть движков оптимизирует этот процесс, но спецификация описывает это именно таким образом. Методы этих объектов также находятся в прототипах, доступных как String.prototype, Number.prototype и Boolean.prototype.

Специальные значения null и undefined не имеют объектов-обёрток, так что методы и свойства им недоступны. Также у них нет соответствующих прототипов.

[**Изменение встроенных прототипов**](https://learn.javascript.ru/native-prototypes#native-prototype-change)

Встроенные прототипы можно изменять. Например, если добавить метод к String.prototype, метод становится доступен для всех строк:

String.prototype.show = function() {

  alert(this);

};

"BOOM!".show(); // BOOM!

В течение процесса разработки могут возникнуть идеи о новых встроенных методах, которые хотелось бы иметь и добавить их во встроенные прототипы. Это плохая идея. Прототипы глобальны, поэтому очень легко могут возникнуть конфликты. Если две библиотеки добавляют метод String.prototype.show, то одна из них перепишет метод другой.

В современном программировании есть только один случай, в котором одобряется изменение встроенных прототипов. Это создание полифилов. Полифил – это термин, который означает замену метода, который существует в спецификации JavaScript, но он ещё не поддерживается текущим движком JavaScript. Тогда можно реализовать его и добавить его во встроенный прототип. Например:

if (!String.prototype.repeat) { // Если такого метода нет

  // добавляем его в прототип

  String.prototype.repeat = function(n) {

    // повторить строку n раз

    return new Array(n + 1).join(this);

  };

}

alert( "La".repeat(3) ); // LaLaLa

[**Заимствование у прототипов**](https://learn.javascript.ru/native-prototypes#zaimstvovanie-u-prototipov)

Ранее рассматривалось заимствовании методов. Это когда метод из одного объекта и копируется в другой. Некоторые методы встроенных прототипов часто одалживают. Например, если создать объект, похожий на массив (псевдомассив), можно скопировать некоторые методы из Array в этот объект. Пример:

let obj = {

  0: "Hello",

  1: "world!",

  length: 2,

};

obj.join = Array.prototype.join;

alert( obj.join(',') ); // Hello,world!

Это работает, потому что для внутреннего алгоритма встроенного метода join важна только корректность индексов и свойства length, он не проверяет является ли объект на самом деле массивом. И многие встроенные методы работают так же. Альтернативная возможность –можно унаследовать от массива, установив obj.\_\_proto\_\_ как Array.prototype, таким образом все методы Array станут автоматически доступны в obj. Но это будет невозможно, если obj уже наследует от другого объекта, ведь можно наследовать только от одного объекта одновременно.

Заимствование методов – гибкий способ, позволяющий смешивать функциональность разных объектов по необходимости.

1. [**Методы прототипов**](https://learn.javascript.ru/prototype-methods)

Ранее упоминалось, что существуют современные методы работы с прототипами. Свойство \_\_proto\_\_ считается устаревшим, и по стандарту должно поддерживаться только браузерами. Современные методы это:

* [Object.create(proto, [descriptors])](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/create) – создаёт пустой объект со свойством [[Prototype]], указанным как proto, и необязательными дескрипторами свойств descriptors.
* [Object.getPrototypeOf(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getPrototypeOf) – возвращает свойство [[Prototype]] объекта obj.
* [Object.setPrototypeOf(obj, proto)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf) – устанавливает свойство [[Prototype]] объекта obj как proto.

Эти методы нужно использовать вместо \_\_proto\_\_. Например:

let animal = {

  eats: true

};

let rabbit = Object.create(animal);

alert(rabbit.eats); // true

alert(Object.getPrototypeOf(rabbit) === animal);

Object.setPrototypeOf(rabbit, {});

У Object.create есть необязательный второй аргумент: дескрипторы свойств. Можно добавить дополнительное свойство новому объекту таким образом:

let animal = {

  eats: true

};

let rabbit = Object.create(animal, {

  jumps: {

    value: true

  }

});

alert(rabbit.jumps); // true

Также можно использовать Object.create для глубокого клонирования объекта, более мощного, чем копирование свойств в цикле for..in:

let clone = Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj));

Такой вызов создаёт точную копию объекта obj, включая все свойства: перечисляемые и не перечисляемые, свойства, геттеры/сеттеры для свойств – и все это с правильным свойством [[Prototype]].

Технически, можно установить/получить [[Prototype]] в любое время. Но обычно прототип устанавливается только раз во время создания объекта, а после не меняется: rabbit наследует от animal, и это не изменится.

JavaScript движки хорошо оптимизированы для этого. Изменение прототипа «на лету» с помощью Object.setPrototypeOf или obj.\_\_proto\_\_= – очень медленная операция, которая ломает внутренние оптимизации для операций доступа к свойствам объекта. Так что лучше избегайте этого, кроме тех случаев, когда знаете, что делаете, либо скорость JavaScript для вас не имеет никакого значения.

Как известно, объекты можно использовать как ассоциативные массивы для хранения пар ключ/значение. Но если попробовать хранить созданные пользователями ключи (например, словари с пользовательским вводом), можно заметить интересный сбой: все ключи работают как ожидается, за исключением "\_\_proto\_\_". Например:

let obj = {};

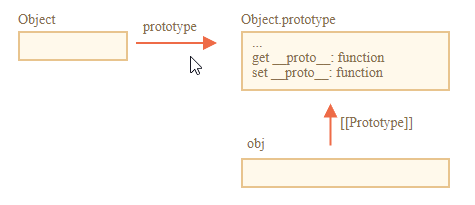
let key = prompt("What's the key?", "\_\_proto\_\_");

obj[key] = "some value";

alert(obj[key]); // [object Object], не "some value"

Если пользователь введёт \_\_proto\_\_, присвоение проигнорируется, так как свойство \_\_proto\_\_ должно быть либо объектом, либо null, а строка не может стать прототипом.

Свойство \_\_proto\_\_ – не обычное, а аксессор, заданный в Object.prototype:



Так что при чтении или установке obj.\_\_proto\_\_ вызывается соответствующий геттер/сеттер из прототипа obj, и именно он устанавливает/получает свойство [[Prototype]].

Поните, что \_\_proto\_\_ – это способ доступа к свойству [[Prototype]], это не само свойство [[Prototype]].

Теперь, если надо использовать объект как ассоциативный массив, можно сделать это следующим образом:

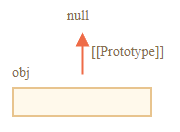
let obj = Object.create(null);

let key = prompt("What's the key?", "\_\_proto\_\_");

obj[key] = "some value";

alert(obj[key]); // "some value"

Object.create(null) создаёт пустой объект без прототипа ([[Prototype]] будет null):



Таким образом не будет унаследованного геттера/сеттера для \_\_proto\_\_. Теперь это свойство обрабатывается как обычное свойство, и приведённый выше пример работает правильно. Можно назвать такой объект «простейшим» или «чистым словарным объектом», потому что он ещё проще чем обычный объект {...}. Недостаток в том, что у таких объектов не будет встроенных методов объекта, таких как toString:

let obj = Object.create(null);

alert(obj); // Error (no toString)

Но обычно это нормально для ассоциативных массивов. Обратите внимание, что большая часть методов, связанных с объектами, имеют вид Object.something(...). К примеру, Object.keys(obj) не находятся в прототипе, так что они продолжат работать для таких объектов:

let chineseDictionary = Object.create(null);

chineseDictionary.hello = "你好";

chineseDictionary.bye = "再见";

alert(Object.keys(chineseDictionary)); // hello,bye

1. [**Классы**](https://learn.javascript.ru/class)

В объектно-ориентированном программировании класс – это расширяемый шаблон кода для создания объектов, который устанавливает в них начальные значения (свойства) и реализацию поведения (методы).

На практике часто надо создавать много объектов одного вида, например пользователей, товары или что-то еще. Как известно, с этим может помочь new function. Но в современном JavaScript есть и более продвинутая конструкция class, которая предоставляет новые возможности, полезные для объектно-ориентированного программирования.

[**Синтаксис «class»**](https://learn.javascript.ru/class#sintaksis-class)

Базовый синтаксис выглядит так:

class MyClass {

  // методы класса

  constructor() { ... }

  method1() { ... }

  method2() { ... }

  method3() { ... }

  ...

}

Затем используйте вызов new MyClass() для создания нового объекта со всеми перечисленными методами. При этом автоматически вызывается метод constructor(), в нём можно инициализовать объект. Например:

class User {

  constructor(name) {

    this.name = name;

  }

  sayHi() {

    alert(this.name);

  }

}

let user = new User("Иван");

user.sayHi();

Когда вызывается new User("Иван"):

1. Создаётся новый объект.
2. constructor запускается с заданным аргументом и сохраняет его в this.name.

Затем можно вызывать методы объекта, такие как user.sayHi().

Методы в классе не разделяются запятой. Это приводит к синтаксической ошибке.

В JavaScript класс – это разновидность функции. Рассмотрим пример:

class User {

  constructor(name) { this.name = name; }

  sayHi() { alert(this.name); }

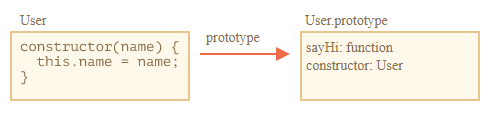
}

alert(typeof User); // function

Вот что на самом деле делает конструкция class User {...}:

1. Создает функцию с именем User, которая становится результатом объявления класса. Код функции берется из метода constructor (она будет пустой, если такого метода нет).
2. Сохраняет все методы, такие как sayHi, в User.prototype.

Затем, при вызове метода на новых объектах new User, он возьмётся из прототипа. Таким образом, объект new User имеет доступ к методам класса. На картинке показан результат объявления class User:



Как видно из кода ниже, класс – это функция или, если точнее, это метод constructor, методы находятся в User.prototype.

class User {

  constructor(name) { this.name = name; }

  sayHi() { alert(this.name); }

}

alert(typeof User); // function

alert(User === User.prototype.constructor); // true

alert(User.prototype.sayHi); // alert(this.name);

alert(Object.getOwnPropertyNames(User.prototype)); // constructor, sayHi

Иногда говорят, что class – это просто «синтаксический сахар» в JavaScript (синтаксис для улучшения читаемости кода, но не делающий ничего принципиально нового), потому что можно сделать все то же самое без конструкции class. Например:

// перепишем класс User с помощью функций

// 1. Создаём функцию constructor

function User(name) {

  this.name = name;

}

// 2. Добавляем метод в прототип

User.prototype.sayHi = function() {

  alert(this.name);

};

let user = new User("Иван");

user.sayHi();

Результат этого кода очень похож на предыдущий. Поэтому, class можно считать синтаксическим сахаром для определения конструктора вместе с методами прототипа. Однако есть важные отличия:

1. Во-первых, функция, созданная с помощью class, помечена специальным внутренним свойством [[FunctionKind]]:"classConstructor". Поэтому это не совсем то же самое, что создавать её вручную.

В отличие от обычных функций, конструктор класса не может быть вызван без new:

class User {

  constructor() {}

}

alert(typeof User); // function

User(); // Error: Class constructor User cannot be invoked without 'new'

Кроме того, строковое представление конструктора класса в большинстве движков JavaScript начинается с «class …».

class User {

  constructor() {}

}

alert(User); // class User { ... }

1. Методы класса являются неперечислимыми. Определение класса устанавливает флаг enumerable в false для всех методов в "prototype".
2. Классы всегда используют use strict. Весь код внутри класса автоматически находится в строгом режиме.

[**Class Expression**](https://learn.javascript.ru/class#class-expression)

Как и функции, классы можно определять внутри другого выражения, передавать, возвращать, присваивать и т.д. Пример Class Expression (по аналогии с Function Expression):

let User = class {

  sayHi() {

    alert("Привет");

  }

};

Как и Named Function Expressions, выражения классов могут иметь имя, которое видно только внутри класса. Если у Class Expression есть имя, то оно видно только внутри класса:

let User = class MyClass {

  sayHi() {

    alert(MyClass);

  }

};

new User().sayHi(); // работает

alert(MyClass); // ошибка

Можно динамически создавать классы «по-запросу»:

function makeClass(phrase) {

  // объявляем класс и возвращаем его

  return class {

    sayHi() {

      alert(phrase);

    };

  };

}

// Создаем новый класс

let User = makeClass("Привет");

new User().sayHi(); // Привет

Как и в литеральных объектах, в классах можно объявлять генераторы, вычисляемые свойства, геттеры/сеттеры и т.д. Пример user.name, реализованного с использованием get/set:

class User {

  constructor(name) {

    // вызывает сеттер

    this.name = name;

  }

  get name() {

    return this.\_name;

  }

  set name(value) {

    if (value.length < 4) {

      alert("Имя слишком короткое.");

      return;

    }

    this.\_name = value;

  }

}

let user = new User("Иван");

alert(user.name); // Иван

user = new User(""); // Имя слишком короткое.

При объявлении класса геттеры/сеттеры создаются на User.prototype:

Object.defineProperties(User.prototype, {

  name: {

    get() {

      return this.\_name

    },

    set(name) {

      // ...

    }

  }

});

Пример с вычисляемым свойством в скобках [...]:

class User {

  ['say' + 'Hi']() {

    alert("Привет");

  }

}

new User().sayHi();

Для методов-генераторов добавьте перед именем \*.

[**Свойства классов**](https://learn.javascript.ru/class#svoystva-klassov)

Свойства классов добавлены в язык недавно. Старым браузерам может понадобиться полифил. В приведённом выше примере у класса User были только методы. Добавим свойство:

class User {

  name = "Аноним";

  sayHi() {

    alert(`Привет, ${this.name}!`);

  }

}

new User().sayHi();

Свойство name не устанавливается в User.prototype. Вместо этого оно создаётся оператором new перед запуском конструктора, это именно свойство объекта.

1. [**Наследование классов**](https://learn.javascript.ru/class-inheritance)

Допустим, у нас есть два класса. Класс Animal:

class Animal {

  constructor(name) {

    this.speed = 0;

    this.name = name;

  }

  run(speed) {

    this.speed += speed;

    alert(`${this.name} бежит со скоростью ${this.speed}.`);

  }

  stop() {

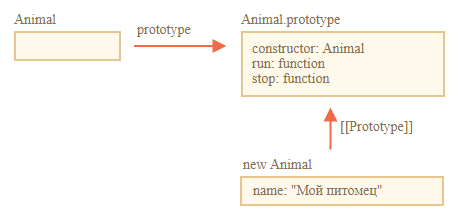
    this.speed = 0;

    alert(`${this.name} стоит.`);

  }

}

let animal = new Animal("Мой питомец");



Класс Rabbit:

class Rabbit {

  constructor(name) {

    this.name = name;

  }

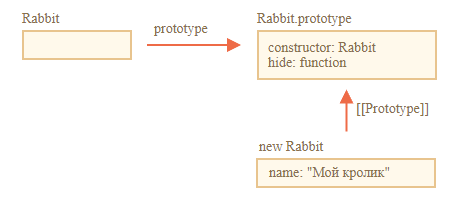
  hide() {

    alert(`${this.name} прячется!`);

  }

}

let rabbit = new Rabbit("Мой кролик");



Для того, чтобы наследовать класс от другого, мы должны использовать ключевое слово "extends" и указать название родительского класса перед {..}. Ниже Rabbit наследует от Animal:

class Animal {

  constructor(name) {

    this.speed = 0;

    this.name = name;

  }

  run(speed) {

    this.speed += speed;

    alert(`${this.name} бежит со скоростью ${this.speed}.`);

  }

  stop() {

    this.speed = 0;

    alert(`${this.name} стоит.`);

  }

}

// Наследуем от Animal указывая "extends Animal"

class Rabbit extends Animal {

  hide() {

    alert(`${this.name} прячется!`);

  }

}

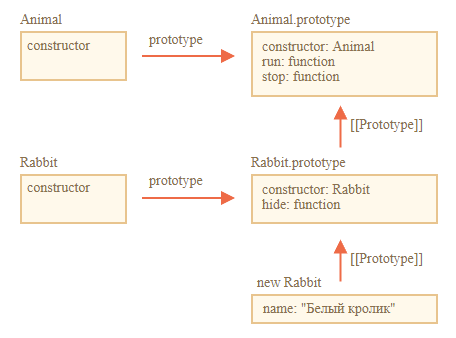
let rabbit = new Rabbit("Белый кролик");

rabbit.run(5); // Белый кролик бежит со скоростью 5.

rabbit.hide(); // Белый кролик прячется!

Теперь код Rabbit стал короче, так как используется конструктор класса Animal по умолчанию и кролик может использовать метод run как и все животные.

На самом деле ключевое слово extends добавляет ссылку на [[Prototype]] из Rabbit.prototype в Animal.prototype:



Если метод не найден в Rabbit.prototype, JavaScript возьмёт его из Animal.prototype.

Синтаксис создания класса допускает указывать после extends не только класс, но любое выражение. Пример вызова функции, которая генерирует родительский класс:

function f(phrase) {

  return class {

    sayHi() { alert(phrase) }

  }

}

class User extends f("Привет") {}

new User().sayHi(); // Привет

Здесь class User наследует от результата вызова f("Привет"). Это может быть полезно для продвинутых приёмов проектирования, где можно использовать функции для генерации классов в зависимости от многих условий и затем наследовать их.

[**Переопределение методов**](https://learn.javascript.ru/class-inheritance#pereopredelenie-metodov)

Сейчас Rabbit наследует от Animal метод stop, который устанавливает this.speed = 0. Если определить свой метод stop в классе Rabbit, то он будет использоваться взамен родительского:

class Rabbit extends Animal {

  stop() {

    // ...будет использован для rabbit.stop()

  }

}

Обычно нет необходимости полностью заменять родительский метод, а только сделать новый на его основе, изменяя или расширяя его функциональность. Для этого надо определить новый метод, добавив нужный функционал, и вызывать родительский метод до/после или в процессе.

У классов есть ключевое слово "super" для таких случаев:

* super.method(...) вызывает родительский метод.
* super(...) вызывает родительский конструктор (работает только внутри нашего конструктора).

Пусть наш кролик автоматически прячется при остановке:

class Animal {

  constructor(name) {

    this.speed = 0;

    this.name = name;

  }

  run(speed) {

    this.speed += speed;

    alert(`${this.name} бежит со скоростью ${this.speed}.`);

  }

  stop() {

    this.speed = 0;

    alert(`${this.name} стоит.`);

  }

}

class Rabbit extends Animal {

  hide() {

    alert(`${this.name} прячется!`);

  }

  stop() {

    super.stop(); // вызываем родительский метод stop

    this.hide(); // и затем hide

  }

}

let rabbit = new Rabbit("Белый кролик");

rabbit.run(5); // Белый кролик бежит со скоростью 5.

rabbit.stop(); // Белый кролик стоит. Белый кролик прячется!

Теперь у класса Rabbit есть метод stop, который вызывает родительский super.stop() в процессе выполнения.

У стрелочных функций нет super. При обращении к super стрелочной функции он берётся из внешней функции:

class Rabbit extends Animal {

  stop() {

    setTimeout(() => super.stop(), 1000); // вызывает родительский stop после 1 секунды

  }

}

В примере super в стрелочной функции тот же самый, что и в stop(), поэтому метод отрабатывает как и ожидается. Если указать здесь «обычную» функцию, была бы ошибка:

// Unexpected super

setTimeout(function() { super.stop() }, 1000);

Согласно [спецификации](https://tc39.github.io/ecma262/#sec-runtime-semantics-classdefinitionevaluation), если класс расширяет другой класс и не имеет конструктора, то автоматически создаётся такой «пустой» конструктор. Например, у Rabbit нет своего конструктора.

class Rabbit extends Animal {

  constructor(...args) {

    super(...args);

  }

}

Как видно, он просто вызывает конструктор родительского класса. Так будет происходить, пока не будет создан собственный конструктор. Добавим конструктор для Rabbit. Он будет устанавливать earLength в дополнение к name:

class Animal {

  constructor(name) {

    this.speed = 0;

    this.name = name;

  }

  // ...

}

class Rabbit extends Animal {

  constructor(name, earLength) {

    this.speed = 0;

    this.name = name;

    this.earLength = earLength;

  }

  // ...

}

let rabbit = new Rabbit("Белый кролик", 10); // Error: this is not defined.

Ошибка возникла из-за того, что в классах-потомках конструктор обязан вызывать super(...) и делать это перед использованием this. Дело в том, что в JavaScript существует различие между «функцией-конструктором наследующего класса» и всеми остальными. В наследующем классе соответствующая функция-конструктор помечена специальным внутренним свойством [[ConstructorKind]]:"derived". Разница в следующем:

* Когда выполняется обычный конструктор, он создаёт пустой объект и присваивает его this.
* Когда запускается конструктор унаследованного класса, он этого не делает. Вместо этого он ждёт, что это сделает конструктор родительского класса.

Поэтому, если создать собственный конструктор, то надо вызвать super, в противном случае объект для this не будет создан, и возникнет ошибка. Чтобы конструктор Rabbit работал, он должен вызвать super() до того, как использовать this, чтобы не было ошибки:

class Animal {

  constructor(name) {

    this.speed = 0;

    this.name = name;

  }

  // ...

}

class Rabbit extends Animal {

  constructor(name, earLength) {

    super(name);

    this.earLength = earLength;

  }

  // ...

}

let rabbit = new Rabbit("Белый кролик", 10);

alert(rabbit.name); // Белый кролик

alert(rabbit.earLength); // 10

[**[[HomeObject]]**](https://learn.javascript.ru/class-inheritance#homeobject)

В JavaScript для функций добавлено специальное внутреннее свойство: [[HomeObject]]. Когда функция объявлена как метод внутри класса или объекта, её свойство [[HomeObject]] становится равно этому объекту. Затем super использует его, чтобы получить прототип родителя и его методы. Посмотрим, как это работает – опять же, используя простые объекты:

let animal = {

  name: "Животное",

  eat() {         // animal.eat.[[HomeObject]] == animal

    alert(`${this.name} eats.`);

  }

};

let rabbit = {

  \_\_proto\_\_: animal,

  name: "Кролик",

  eat() {         // rabbit.eat.[[HomeObject]] == rabbit

    super.eat();

  }

};

let longEar = {

  \_\_proto\_\_: rabbit,

  name: "Длинноух",

  eat() {         // longEar.eat.[[HomeObject]] == longEar

    super.eat();

  }

};

longEar.eat();  // Длинноух ест.

Это работает как задумано благодаря [[HomeObject]]. Метод, такой как longEar.eat, знает свой [[HomeObject]] и получает метод родителя из его прототипа. Вообще без использования this.

[**Методы не «свободны»**](https://learn.javascript.ru/class-inheritance#metody-ne-svobodny)

Как известно, функции в JavaScript «свободны», не привязаны к объектам. Их можно копировать между объектами и вызывать с любым this. Но само существование [[HomeObject]] нарушает этот принцип, так как методы запоминают свои объекты. [[HomeObject]] нельзя изменить, эта связь – навсегда. Единственное место в языке, где используется [[HomeObject]] – это super. Поэтому если метод не использует super, то все ещё можно считать его свободным и копировать между объектами. А вот если super в коде есть, то возможны побочные эффекты. Вот пример неверного результата super после копирования:

let animal = {

  sayHi() {

    console.log(`Я животное`);

  }

};

// rabbit наследует от animal

let rabbit = {

  \_\_proto\_\_: animal,

  sayHi() {

    super.sayHi();

  }

};

let plant = {

  sayHi() {

    console.log("Я растение");

  }

};

// tree наследует от plant

let tree = {

  \_\_proto\_\_: plant,

  sayHi: rabbit.sayHi // (\*)

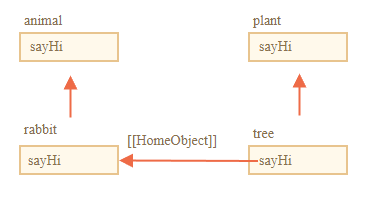
};

tree.sayHi();  // Я животное

Вызов tree.sayHi() показывает «Я животное». Определённо неверно. Причина проста:

* В строке (\*), метод tree.sayHi скопирован из rabbit.
* Его [[HomeObject]] – это rabbit, ведь он был создан в rabbit. Свойство [[HomeObject]] никогда не меняется.
* В коде tree.sayHi() есть вызов super.sayHi(). Он идёт вверх от rabbit и берёт метод из animal.

Вот диаграмма происходящего:



Свойство [[HomeObject]] определено для методов как классов, так и обычных объектов. Но для объектов методы должны быть объявлены именно как method(), а не "method: function()". В приведённом ниже примере используется синтаксис не метода, свойства-функции. Поэтому у него нет [[HomeObject]], и наследование не работает:

let animal = {

  eat: function() { // должен быть короткий синтаксис: eat() {...}

    // ...

  }

};

let rabbit = {

  \_\_proto\_\_: animal,

  eat: function() {

    super.eat();

  }

};

rabbit.eat();  // Ошибка вызова super

1. [**Статические свойства и методы**](https://learn.javascript.ru/static-properties-methods)

Можно присвоить метод самой функции-классу, а не её "prototype". Такие методы называются статическими. В классе такие методы обозначаются ключевым словом static, например:

class User {

  static staticMethod() {

    alert(this === User);

  }

}

User.staticMethod(); // true

Это фактически то же самое, что присвоить метод напрямую как свойство функции:

class User() { }

User.staticMethod = function() {

  alert(this === User);

};

Значением this при вызове User.staticMethod() является сам конструктор класса User (правило «объект до точки»).

Обычно статические методы используются для реализации функций, принадлежащих классу, но не к каким-то конкретным его объектам. Например, есть объекты статей Article, и нужна функция для их сравнения. Естественное решение – сделать для этого метод Article.compare:

class Article {

  constructor(title, date) {

    this.title = title;

    this.date = date;

  }

  static compare(articleA, articleB) {

    return articleA.date - articleB.date;

  }

}

let articles = [

  new Article("HTML", new Date(2019, 1, 1)),

  new Article("CSS", new Date(2019, 0, 1)),

  new Article("JavaScript", new Date(2019, 11, 1))

];

articles.sort(Article.compare);

alert( articles[0].title ); // CSS

Здесь метод Article.compare стоит «над» статьями, как способ их сравнения. Это метод не отдельной статьи, а всего класса. Другим примером может быть так называемый «фабричный» метод. Представим, что нужно создавать статьи различными способами:

1. Создание через заданные параметры (title, date и т. д.).
2. Создание пустой статьи с сегодняшней датой и др.

Первый способ может быть реализован через конструктор. А для второго можно использовать статический метод класса. Такой как Article.createTodays() в следующем примере:

class Article {

  constructor(title, date) {

    this.title = title;

    this.date = date;

  }

  static createTodays() {

    // this = Article

    return new this("Сегодняшний дайджест", new Date());

  }

}

let article = Article.createTodays();

alert( article.title ); // Сегодняшний дайджест

Теперь каждый раз, когда нужно создать сегодняшний дайджест, нужно вызывать Article.createTodays(). Ещё раз, это не метод одной статьи, а метод всего класса. Статические методы также используются в классах, относящихся к базам данных, для поиска/сохранения/удаления вхождений в базу данных. Например, предположим, что Article - это специальный класс для управления статьями статический метод для удаления статьи:

Article.remove({id: 12345});

[**Статические свойства**](https://learn.javascript.ru/static-properties-methods#staticheskie-svoystva)

Эта возможность была добавлена в язык недавно. Примеры работают в последнем Chrome. Статические свойства также возможны, они выглядят как свойства класса, но с static в начале:

class Article {

  static publisher = "Иван Иванов";

}

alert( Article.publisher ); // Иван Иванов

Это то же самое, что и прямое присваивание Article:

Article.publisher = "Иван Иванов";

Статические свойства и методы наследуются. Например, метод Animal.compare в коде ниже наследуется и доступен как Rabbit.compare:

class Animal {

  constructor(name, speed) {

    this.speed = speed;

    this.name = name;

  }

  run(speed = 0) {

    this.speed += speed;

    alert(`${this.name} бежит со скоростью ${this.speed}.`);

  }

  static compare(animalA, animalB) {

    return animalA.speed - animalB.speed;

  }

}

// Наследует от Animal

class Rabbit extends Animal {

  hide() {

    alert(`${this.name} прячется!`);

  }

}

let rabbits = [

  new Rabbit("Белый кролик", 10),

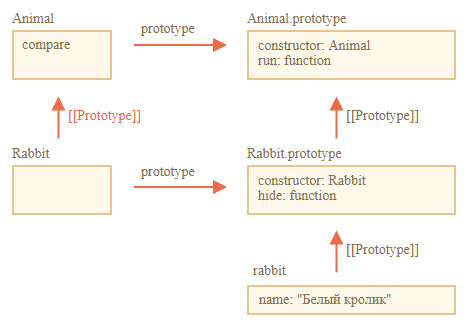
  new Rabbit("Чёрный кролик", 5)

];

rabbits.sort(Rabbit.compare);

rabbits[0].run(); // Чёрный кролик бежит со скоростью 5.

Можно вызвать Rabbit.compare, при этом будет вызван унаследованный Animal.compare. Это работает с использованием прототипов. Extends даёт Rabbit ссылку [[Prototype]] на Animal.



Так что Rabbit extends Animal создаёт две ссылки на прототип:

1. Функция Rabbit прототипно наследует от Animal function.
2. Rabbit.prototype прототипно наследует от Animal.prototype.

В результате наследование работает как для обычных, так и для статических методов:

class Animal {}

class Rabbit extends Animal {}

// для статики

alert(Rabbit.\_\_proto\_\_ === Animal); // true

// для обычных методов

alert(Rabbit.prototype.\_\_proto\_\_ === Animal.prototype);

1. [**Приватные и защищённые методы и свойства**](https://learn.javascript.ru/private-protected-properties-methods)

В объектно-ориентированном программировании свойства и методы разделены на две группы:

* *Внутренний интерфейс* – методы и свойства, доступные из других методов класса, но не снаружи класса.
* *Внешний интерфейс* – методы и свойства, доступные снаружи класса.

Внутренний интерфейс используется для работы объекта, его методы, свойства используют друг друга.  Всё, что нужно для использования объекта, это знать его внешний интерфейс. Совершенно не обязательно знать, его внутреннюю структуру и логику.

В JavaScript есть два типа полей (свойств и методов) объекта:

* *Публичные*: доступны отовсюду. Они составляют внешний интерфейс. До этого момента в рассматриваемых примерах использовались только публичные свойства и методы.
* *Приватные*: доступны только внутри класса. Они для внутреннего интерфейса.

Во многих других языках также существуют «защищённые» поля, доступные только внутри класса или для дочерних классов (то есть, как приватные, но разрёшен доступ для наследующих классов) и также полезны для внутреннего интерфейса. В некотором смысле они более распространены, чем приватные, потому что обычно надо, чтобы наследующие классы получали доступ к внутренним полям.

Защищённые поля не реализованы в JavaScript на уровне языка, но на практике они очень удобны, поэтому их эмулируют.

Создадим простой класс для описания работы кофеварки:

class CoffeeMachine {

  waterAmount = 0; // количество воды внутри

  constructor(power) {

    this.power = power;

    alert( `Создана кофеварка, мощность: ${power}` );

  }

}

// создаём кофеварку

let coffeeMachine = new CoffeeMachine(100);

// добавляем воды

coffeeMachine.waterAmount = 200;

Сейчас свойства waterAmount и power публичные. Можно легко получать и устанавливать им любое значение извне. Изменим свойство waterAmount на защищённое, чтобы иметь больше контроля над ним. Например, не надо, чтобы кто-либо устанавливал его ниже нуля.

Защищённые свойства обычно начинаются с префикса \_. Это не синтаксис языка: есть хорошо известное соглашение между программистами, что такие свойства и методы не должны быть доступны извне. Большинство программистов следуют этому соглашению. Пожтому свойство будет называться \_waterAmount:

class CoffeeMachine {

  \_waterAmount = 0;

  set waterAmount(value) {

    if (value < 0) throw new Error("Отрицательное количество воды");

    this.\_waterAmount = value;

  }

  get waterAmount() {

    return this.\_waterAmount;

  }

  constructor(power) {

    this.\_power = power;

  }

}

// создаём новую кофеварку

let coffeeMachine = new CoffeeMachine(100);

// устанавливаем количество воды

coffeeMachine.waterAmount = -10; // Error: Отрицательное количество воды

Теперь доступ под контролем, поэтому указать воду ниже нуля не удалось.

Сделаем свойство power доступным только для чтения. Иногда нужно, чтобы свойство устанавливалось только при создании объекта и после этого никогда не изменялось. Это как раз требуется для кофеварки: мощность никогда не меняется. Для этого нужно создать только геттер, но не сеттер:

class CoffeeMachine {

  // ...

  constructor(power) {

    this.\_power = power;

  }

  get power() {

    return this.\_power;

  }

}

let coffeeMachine = new CoffeeMachine(100);

alert(`Мощность: ${coffeeMachine.power}W`); // Мощность: 100W

coffeeMachine.power = 25; // Error (no setter)

Здесь использовался синтаксис геттеров/сеттеров. Но в большинстве случаев использование функций get.../set... предпочтительнее:

class CoffeeMachine {

  \_waterAmount = 0;

  setWaterAmount(value) {

    if (value < 0) throw new Error("Отрицательное количество воды");

    this.\_waterAmount = value;

  }

  getWaterAmount() {

    return this.\_waterAmount;

  }

}

new CoffeeMachine().setWaterAmount(100);

Это выглядит немного длиннее, но функции более гибкие. Они могут принимать несколько аргументов.

Если унаследовать class MegaMachine extends CoffeeMachine, ничто не помешает нам обращаться к this.\_waterAmount или this.\_power из методов нового класса. Таким образом защищённые методы, конечно же, наследуются. В отличие от приватных полей.

[**Приватное свойство «#waterLimit»**](https://learn.javascript.ru/private-protected-properties-methods#privatnoe-svoystvo-waterlimit)

Поддержка приватных свойств и методов была добавлена в язык недавно. В движках JavaScript пока не поддерживается или поддерживается частично, нужен полифилл. Приватные свойства и методы должны начинаться со знака #. Они доступны только внутри класса. Например, в классе ниже есть приватное свойство #waterLimit и приватный метод #checkWater для проверки количества воды:

class CoffeeMachine {

  #waterLimit = 200;

  #checkWater(value) {

    if (value < 0) throw new Error("Отрицательный уровень воды");

    if (value > this.#waterLimit) throw new Error("Слишком много воды");

  }

}

let coffeeMachine = new CoffeeMachine();

// снаружи нет доступа к приватным методам класса

coffeeMachine.#checkWater(); // Error

coffeeMachine.#waterLimit = 1000; // Error

На уровне языка # является специальным символом, который означает, что поле приватное. Нельзя получить к нему доступ извне или из наследуемых классов. Приватные поля не конфликтуют с публичными. Может быть два поля одновременно – приватное #waterAmount и публичное waterAmount.

Например, давайте сделаем аксессор waterAmount для #waterAmount:

class CoffeeMachine {

  #waterAmount = 0;

  get waterAmount() {

    return this.#waterAmount;

  }

  set waterAmount(value) {

    if (value < 0) throw new Error("Отрицательный уровень воды");

    this.#waterAmount = value;

  }

}

let machine = new CoffeeMachine();

machine.waterAmount = 100;

alert(machine.#waterAmount); // Error

В отличие от защищённых, функционал приватных полей обеспечивается самим языком. Но если унаследовать от CoffeeMachine, то не будет прямого доступа к #waterAmount, только через геттер/сеттер waterAmount:

class MegaCoffeeMachine extends CoffeeMachine() {

  method() {

    alert( this.#waterAmount ); // Error: can only access from CoffeeMachine

  }

}

Во многих случаях такое ограничение неудобно. Если расширяется CoffeeMachine, то должен быть доступ к внутренним методам и свойствам. Поэтому защищённые свойства используются чаще, хоть они и не поддерживаются синтаксисом языка.

Как известно, обычно можно получить доступ к полям объекта с помощью this[name]:

class User {

  ...

  sayHi() {

    let fieldName = "name";

    alert(`Hello, ${this[fieldName]}`);

  }

}

С приватными свойствами такое невозможно: this['#name'] не работает. Это ограничение синтаксиса сделано для обеспечения приватности.

1. [**Статические свойства и методы**](https://learn.javascript.ru/static-properties-methods)

Можно присвоить метод самой функции-классу, а не её "prototype". Такие методы называются статическими. В классе такие методы обозначаются ключевым словом static, например:

class User {

  static staticMethod() {

    alert(this === User);

  }

}

User.staticMethod(); // true

Это фактически то же самое, что присвоить метод напрямую как свойство функции:

class User() { }

User.staticMethod = function() {

  alert(this === User);

};

Значением this при вызове User.staticMethod() является сам конструктор класса User (правило «объект до точки»).

Обычно статические методы используются для реализации функций, принадлежащих классу, но не к каким-то конкретным его объектам. Например, есть объекты статей Article, и нужна функция для их сравнения. Естественное решение – сделать для этого метод Article.compare:

class Article {

  constructor(title, date) {

    this.title = title;

    this.date = date;

  }

  static compare(articleA, articleB) {

    return articleA.date - articleB.date;

  }

}

let articles = [

  new Article("HTML", new Date(2019, 1, 1)),

  new Article("CSS", new Date(2019, 0, 1)),

  new Article("JavaScript", new Date(2019, 11, 1))

];

articles.sort(Article.compare);

alert( articles[0].title ); // CSS

Здесь метод Article.compare стоит «над» статьями, как способ их сравнения. Это метод не отдельной статьи, а всего класса. Другим примером может быть так называемый «фабричный» метод. Представим, что нужно создавать статьи различными способами:

1. Создание через заданные параметры (title, date и т. д.).
2. Создание пустой статьи с сегодняшней датой и др.

Первый способ может быть реализован через конструктор. А для второго можно использовать статический метод класса. Такой как Article.createTodays() в следующем примере:

class Article {

  constructor(title, date) {

    this.title = title;

    this.date = date;

  }

  static createTodays() {

    // this = Article

    return new this("Сегодняшний дайджест", new Date());

  }

}

let article = Article.createTodays();

alert( article.title ); // Сегодняшний дайджест

Теперь каждый раз, когда нужно создать сегодняшний дайджест, нужно вызывать Article.createTodays(). Ещё раз, это не метод одной статьи, а метод всего класса. Статические методы также используются в классах, относящихся к базам данных, для поиска/сохранения/удаления вхождений в базу данных. Например, предположим, что Article - это специальный класс для управления статьями статический метод для удаления статьи:

Article.remove({id: 12345});

[**Статические свойства**](https://learn.javascript.ru/static-properties-methods#staticheskie-svoystva)

Эта возможность была добавлена в язык недавно. Примеры работают в последнем Chrome. Статические свойства также возможны, они выглядят как свойства класса, но с static в начале:

class Article {

  static publisher = "Иван Иванов";

}

alert( Article.publisher ); // Иван Иванов

Это то же самое, что и прямое присваивание Article:

Article.publisher = "Иван Иванов";

Статические свойства и методы наследуются. Например, метод Animal.compare в коде ниже наследуется и доступен как Rabbit.compare:

class Animal {

  constructor(name, speed) {

    this.speed = speed;

    this.name = name;

  }

  run(speed = 0) {

    this.speed += speed;

    alert(`${this.name} бежит со скоростью ${this.speed}.`);

  }

  static compare(animalA, animalB) {

    return animalA.speed - animalB.speed;

  }

}

// Наследует от Animal

class Rabbit extends Animal {

  hide() {

    alert(`${this.name} прячется!`);

  }

}

let rabbits = [

  new Rabbit("Белый кролик", 10),

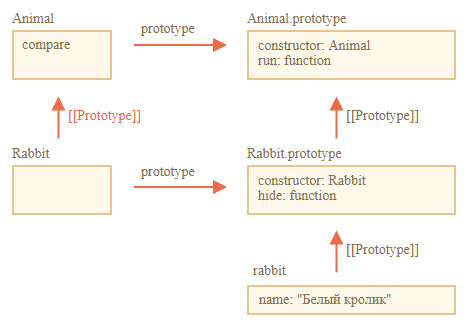
  new Rabbit("Чёрный кролик", 5)

];

rabbits.sort(Rabbit.compare);

rabbits[0].run(); // Чёрный кролик бежит со скоростью 5.

Можно вызвать Rabbit.compare, при этом будет вызван унаследованный Animal.compare. Это работает с использованием прототипов. Extends даёт Rabbit ссылку [[Prototype]] на Animal.



Так что Rabbit extends Animal создаёт две ссылки на прототип:

1. Функция Rabbit прототипно наследует от Animal function.
2. Rabbit.prototype прототипно наследует от Animal.prototype.

В результате наследование работает как для обычных, так и для статических методов:

class Animal {}

class Rabbit extends Animal {}

// для статики

alert(Rabbit.\_\_proto\_\_ === Animal); // true

// для обычных методов

alert(Rabbit.prototype.\_\_proto\_\_ === Animal.prototype);

1. [**Приватные и защищённые методы и свойства**](https://learn.javascript.ru/private-protected-properties-methods)

В объектно-ориентированном программировании свойства и методы разделены на две группы:

* *Внутренний интерфейс* – методы и свойства, доступные из других методов класса, но не снаружи класса.
* *Внешний интерфейс* – методы и свойства, доступные снаружи класса.

Внутренний интерфейс используется для работы объекта, его методы, свойства используют друг друга.  Всё, что нужно для использования объекта, это знать его внешний интерфейс. Совершенно не обязательно знать, его внутреннюю структуру и логику.

В JavaScript есть два типа полей (свойств и методов) объекта:

* *Публичные*: доступны отовсюду. Они составляют внешний интерфейс. До этого момента в рассматриваемых примерах использовались только публичные свойства и методы.
* *Приватные*: доступны только внутри класса. Они для внутреннего интерфейса.

Во многих других языках также существуют «защищённые» поля, доступные только внутри класса или для дочерних классов (то есть, как приватные, но разрёшен доступ для наследующих классов) и также полезны для внутреннего интерфейса. В некотором смысле они более распространены, чем приватные, потому что обычно надо, чтобы наследующие классы получали доступ к внутренним полям.

Защищённые поля не реализованы в JavaScript на уровне языка, но на практике они очень удобны, поэтому их эмулируют.

Создадим простой класс для описания работы кофеварки:

class CoffeeMachine {

  waterAmount = 0; // количество воды внутри

  constructor(power) {

    this.power = power;

    alert( `Создана кофеварка, мощность: ${power}` );

  }

}

// создаём кофеварку

let coffeeMachine = new CoffeeMachine(100);

// добавляем воды

coffeeMachine.waterAmount = 200;

Сейчас свойства waterAmount и power публичные. Можно легко получать и устанавливать им любое значение извне. Изменим свойство waterAmount на защищённое, чтобы иметь больше контроля над ним. Например, не надо, чтобы кто-либо устанавливал его ниже нуля.

Защищённые свойства обычно начинаются с префикса \_. Это не синтаксис языка: есть хорошо известное соглашение между программистами, что такие свойства и методы не должны быть доступны извне. Большинство программистов следуют этому соглашению. Пожтому свойство будет называться \_waterAmount:

class CoffeeMachine {

  \_waterAmount = 0;

  set waterAmount(value) {

    if (value < 0) throw new Error("Отрицательное количество воды");

    this.\_waterAmount = value;

  }

  get waterAmount() {

    return this.\_waterAmount;

  }

  constructor(power) {

    this.\_power = power;

  }

}

// создаём новую кофеварку

let coffeeMachine = new CoffeeMachine(100);

// устанавливаем количество воды

coffeeMachine.waterAmount = -10; // Error: Отрицательное количество воды

Теперь доступ под контролем, поэтому указать воду ниже нуля не удалось.

Сделаем свойство power доступным только для чтения. Иногда нужно, чтобы свойство устанавливалось только при создании объекта и после этого никогда не изменялось. Это как раз требуется для кофеварки: мощность никогда не меняется. Для этого нужно создать только геттер, но не сеттер:

class CoffeeMachine {

  // ...

  constructor(power) {

    this.\_power = power;

  }

  get power() {

    return this.\_power;

  }

}

let coffeeMachine = new CoffeeMachine(100);

alert(`Мощность: ${coffeeMachine.power}W`); // Мощность: 100W

coffeeMachine.power = 25; // Error (no setter)

Здесь использовался синтаксис геттеров/сеттеров. Но в большинстве случаев использование функций get.../set... предпочтительнее:

class CoffeeMachine {

  \_waterAmount = 0;

  setWaterAmount(value) {

    if (value < 0) throw new Error("Отрицательное количество воды");

    this.\_waterAmount = value;

  }

  getWaterAmount() {

    return this.\_waterAmount;

  }

}

new CoffeeMachine().setWaterAmount(100);

Это выглядит немного длиннее, но функции более гибкие. Они могут принимать несколько аргументов.

Если унаследовать class MegaMachine extends CoffeeMachine, ничто не помешает нам обращаться к this.\_waterAmount или this.\_power из методов нового класса. Таким образом защищённые методы, конечно же, наследуются. В отличие от приватных полей.

[**Приватное свойство «#waterLimit»**](https://learn.javascript.ru/private-protected-properties-methods#privatnoe-svoystvo-waterlimit)

Поддержка приватных свойств и методов была добавлена в язык недавно. В движках JavaScript пока не поддерживается или поддерживается частично, нужен полифилл. Приватные свойства и методы должны начинаться со знака #. Они доступны только внутри класса. Например, в классе ниже есть приватное свойство #waterLimit и приватный метод #checkWater для проверки количества воды:

class CoffeeMachine {

  #waterLimit = 200;

  #checkWater(value) {

    if (value < 0) throw new Error("Отрицательный уровень воды");

    if (value > this.#waterLimit) throw new Error("Слишком много воды");

  }

}

let coffeeMachine = new CoffeeMachine();

// снаружи нет доступа к приватным методам класса

coffeeMachine.#checkWater(); // Error

coffeeMachine.#waterLimit = 1000; // Error

На уровне языка # является специальным символом, который означает, что поле приватное. Нельзя получить к нему доступ извне или из наследуемых классов. Приватные поля не конфликтуют с публичными. Может быть два поля одновременно – приватное #waterAmount и публичное waterAmount.

Например, давайте сделаем аксессор waterAmount для #waterAmount:

class CoffeeMachine {

  #waterAmount = 0;

  get waterAmount() {

    return this.#waterAmount;

  }

  set waterAmount(value) {

    if (value < 0) throw new Error("Отрицательный уровень воды");

    this.#waterAmount = value;

  }

}

let machine = new CoffeeMachine();

machine.waterAmount = 100;

alert(machine.#waterAmount); // Error

В отличие от защищённых, функционал приватных полей обеспечивается самим языком. Но если унаследовать от CoffeeMachine, то не будет прямого доступа к #waterAmount, только через геттер/сеттер waterAmount:

class MegaCoffeeMachine extends CoffeeMachine() {

  method() {

    alert( this.#waterAmount ); // Error: can only access from CoffeeMachine

  }

}

Во многих случаях такое ограничение неудобно. Если расширяется CoffeeMachine, то должен быть доступ к внутренним методам и свойствам. Поэтому защищённые свойства используются чаще, хоть они и не поддерживаются синтаксисом языка.

Как известно, обычно можно получить доступ к полям объекта с помощью this[name]:

class User {

  ...

  sayHi() {

    let fieldName = "name";

    alert(`Hello, ${this[fieldName]}`);

  }

}

С приватными свойствами такое невозможно: this['#name'] не работает. Это ограничение синтаксиса сделано для обеспечения приватности.

1. [**Расширение встроенных классов**](https://learn.javascript.ru/extend-natives)

От встроенных классов, таких как Array, Map и других, тоже можно наследовать. Например, в этом примере PowerArray наследуется от встроенного Array:

class PowerArray extends Array {

  isEmpty() {

    return this.length === 0;

  }

}

let arr = new PowerArray(1, 2, 5, 10, 50);

alert(arr.isEmpty()); // false

let filteredArr = arr.filter(item => item >= 10);

alert(filteredArr); // 10, 50

alert(filteredArr.isEmpty()); // false

Обратите внимание: встроенные методы, такие как filter, map и другие возвращают новые объекты унаследованного класса PowerArray. Их внутренняя реализация такова, что для этого они используют свойство объекта constructor.

В примере выше arr.constructor === PowerArray. Поэтому при вызове метода arr.filter() он внутри создаёт массив результатов, именно используя arr.constructor, а не обычный массив. Это удобно, поскольку можно продолжать использовать методы PowerArray далее на результатах.

Более того, можно настроить это поведение. При помощи специального статического геттера Symbol.species можно вернуть конструктор, который JavaScript будет использовать в filter, map и других методах для создания новых объектов. Если надо, чтобы методы map, filter и т. д. возвращали обычные массивы, то нужно вернуть Array в Symbol.species, вот так:

class PowerArray extends Array {

  isEmpty() {

    return this.length === 0;

  }

  // встроенные методы массива будут использовать этот метод как конструктор

  static get [Symbol.species]() {

    return Array;

  }

}

let arr = new PowerArray(1, 2, 5, 10, 50);

alert(arr.isEmpty()); // false

// filter создаст новый массив, используя arr.constructor[Symbol.species] как конструктор

let filteredArr = arr.filter(item => item >= 10);

// filteredArr не является PowerArray, это Array

alert(filteredArr.isEmpty()); // Error: filteredArr.isEmpty is not a function

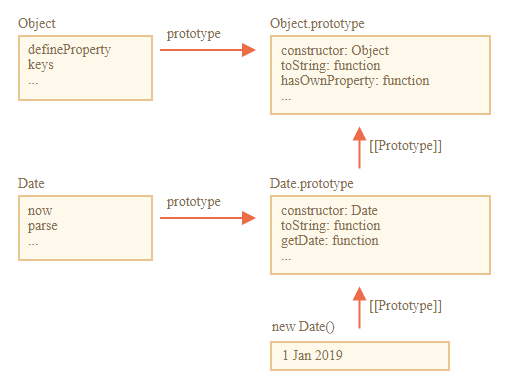
Теперь .filter возвращает Array. Расширенная функциональность не будет передаваться далее.

Другие коллекции, такие как Map, Set, работают аналогично. Они также исполуют Symbol.species.

[**Отсутствие статического наследования встроенных классов**](https://learn.javascript.ru/extend-natives#otsutstvie-staticheskogo-nasledovaniya-vstroennyh-klassov)

У встроенных объектов есть собственные статические методы, например, Object.keys, Array.isArray и т. д. Как известно, встроенные классы расширяют друг друга. Обычно, когда один класс наследует другому, то наследуются и статические методы. Но встроенные классы – исключение. Они не наследуют статические методы друг друуга. Например, и Array и Date наследуют от Object, так что в их экземплярах доступны методы из Object.prototype. Но Array.[[Prototype]] не ссылается на Object, поэтому нет методов Array.keys() или Date.keys().

Ниже представлена структура Date и Object:



Как видно, нет связи между Date и Object. Они независимы, только Date.prototype наследует от Object.prototype. В этом важное отличие наследования встроенных объектов от тех, которые получаются с использованием extends.