**Тема 5. ООП в функциональном и прототипном стиле**

* 1. [Прототипное наследование](https://learn.javascript.ru/prototype-inheritance).
  2. [F.prototype](https://learn.javascript.ru/function-prototype).
  3. [Встроенные прототипы](https://learn.javascript.ru/native-prototypes).
  4. [Методы прототипов.](https://learn.javascript.ru/prototype-methods)
  5. [Классы.](https://learn.javascript.ru/class)
  6. [Наследование классов](https://learn.javascript.ru/class-inheritance).
  7. [Статические свойства и методы](https://learn.javascript.ru/static-properties-methods).
  8. [Приватные и защищённые методы и свойства](https://learn.javascript.ru/private-protected-properties-methods).
  9. [Расширение встроенных классов](https://learn.javascript.ru/extend-natives).
  10. [Проверка класса: "instanceof"](https://learn.javascript.ru/instanceof).
  11. [Примеси](https://learn.javascript.ru/mixins).
  12. Обработка ошибок, "try..catch".
  13. Пользовательские ошибки, расширение Error.
  14. Модули: введение.
  15. Модули: экспорт и импорт.
  16. Модули: динамические импорты.

Содержание данной темы включает материалы, доступные по адресу https://learn.javascript.ru.

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

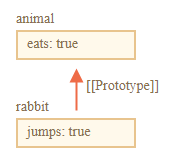
};

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 = {

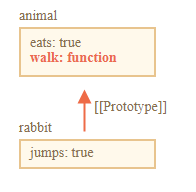
jumps: true,

\_\_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 = {

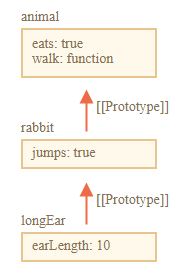
earLength: 10,

\_\_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

};

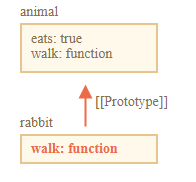
rabbit.walk = function() {

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

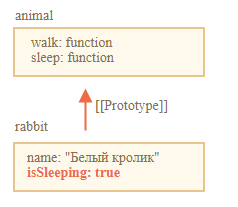
};

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

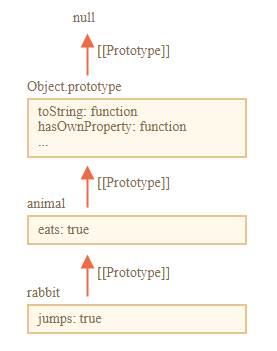
} else {

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;

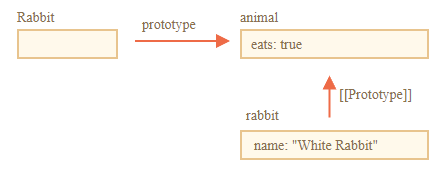
}

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.

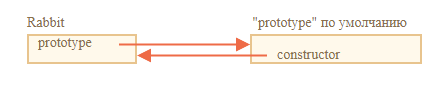
У каждой функции по умолчанию уже есть свойство "prototype". По умолчанию "prototype" – объект с единственным свойством constructor, которое ссылается на функцию-конструктор. Вот такой:

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]]:

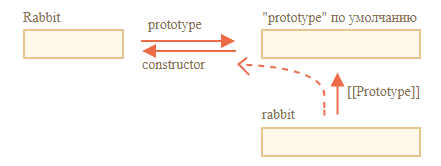
function Rabbit() {}

// 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 внутри самого языка JavaScript. Все встроенные функции-конструкторы используют его.

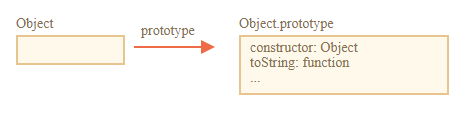
[**Object.prototype**](https://learn.javascript.ru/native-prototypes#object-prototype)

Выведем пустой объект:

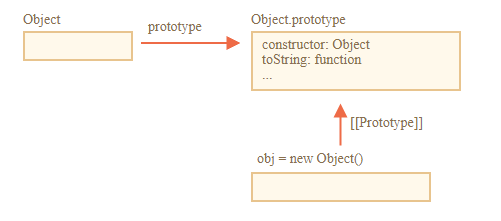
let obj = {};

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

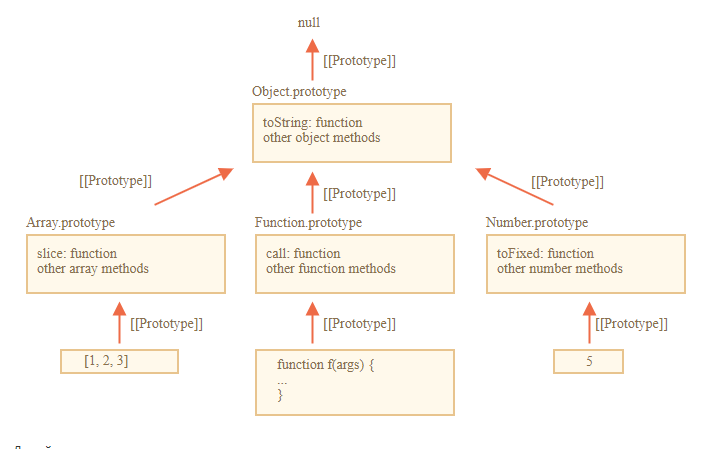
Обратим внимание, что выше Object.prototype по цепочке прототипов больше нет [[Prototype]]:

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

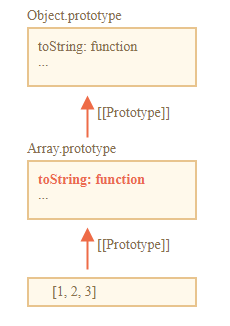
alert( arr.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_ ); // null

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

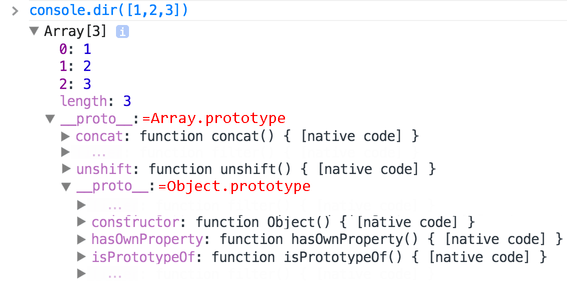
let arr = [1, 2, 3]

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 = {};

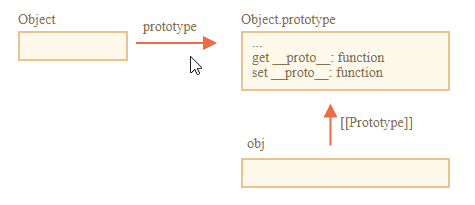
let key = prompt("What's the key?", "\_\_proto\_\_");

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:

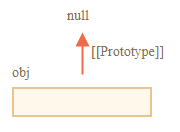
let obj = Object.create(null);

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); }

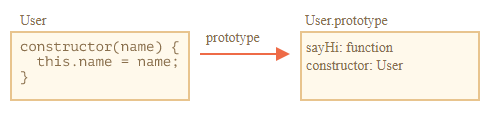
}

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() {

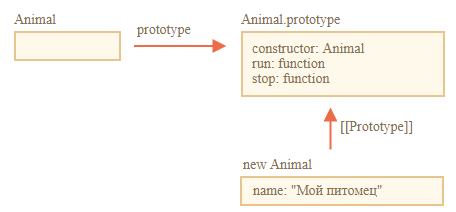
this.speed = 0;

alert(`${this.name} стоит.`);

}

}

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



Класс Rabbit:

class Rabbit {

constructor(name) {

this.name = name;

}

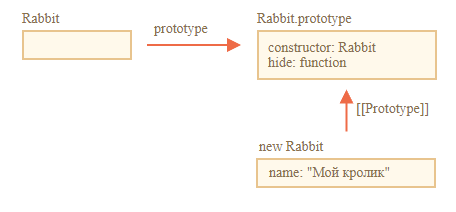
hide() {

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!`);

}

}

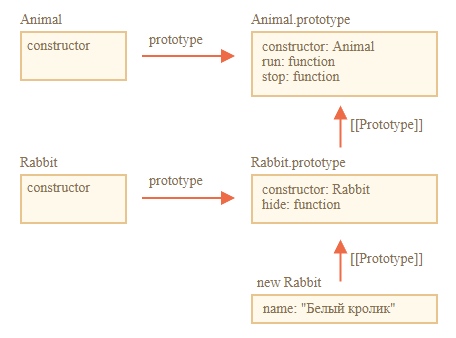
let rabbit = new Rabbit("Белый кролик");

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" \l "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 // (\*)

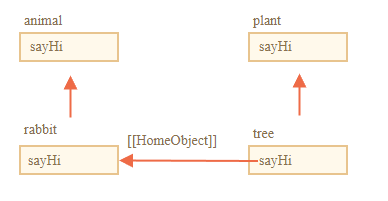
};

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),

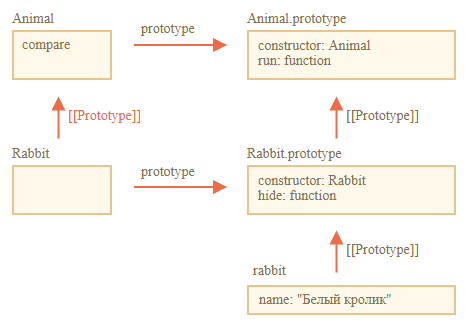
new Rabbit("Чёрный кролик", 5)

];

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" \l "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

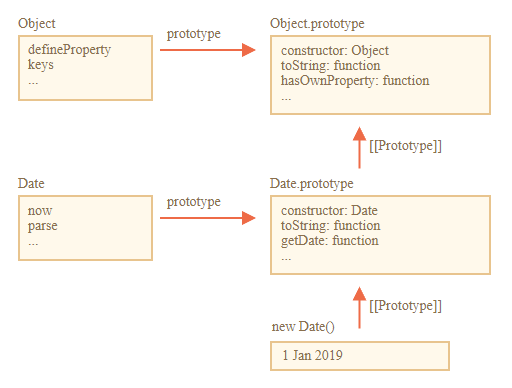
Теперь .filter возвращает Array. Расширенная функциональность не будет передаваться далее.

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.

1. [**Проверка класса: "instanceof"**](https://learn.javascript.ru/instanceof)

Оператор instanceof позволяет проверить, к какому классу принадлежит объект, с учётом наследования. Такая проверка может потребоваться во многих случаях. В рассматриваемых примерах она используется для создания полиморфной функции, которая интерпретирует аргументы по-разному в зависимости от их типа.

[**Оператор instanceof**](https://learn.javascript.ru/instanceof#ref-instanceof)

Синтаксис:

obj instanceof Class

Оператор вернёт true, если obj принадлежит классу Class или наследующему от него. Например:

class Rabbit {}

let rabbit = new Rabbit();

alert( rabbit instanceof Rabbit ); // true

Также это работает с функциями-конструкторами:

function Rabbit() {}

alert( new Rabbit() instanceof Rabbit ); // true

И для встроенных классов, таких как Array:

let arr = [1, 2, 3];

alert( arr instanceof Array ); // true

alert( arr instanceof Object ); // true

Обратите внимание, что arr также принадлежит классу Object, потому что Array наследует от Object.

Обычно оператор instanceof просматривает для проверки цепочку прототипов. Но это поведение может быть изменено при помощи статического метода Symbol.hasInstance.

Алгоритм работы obj instanceof Class работает примерно так:

1. Если имеется статический метод Symbol.hasInstance, тогда вызвать его: Class[Symbol.hasInstance](obj). Он должен вернуть либо true, либо false. Это как раз и есть возможность ручной настройки instanceof.

Пример:

class Animal {

static [Symbol.hasInstance](obj) {

if (obj.canEat) return true;

}

}

let obj = { canEat: true };

alert(obj instanceof Animal); // true: вызван Animal[Symbol.hasInstance](obj)

1. Большая часть классов не имеет метода Symbol.hasInstance. В этом случае используется стандартная логика: проверяется, равен ли Class.prototype одному из прототипов в прототипной цепочке obj.

Другими словами, сравнивается:

obj.\_\_proto\_\_ === Class.prototype?

obj.\_\_proto\_\_.\_\_proto\_\_ === Class.prototype?

obj.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_ === Class.prototype?

...

// если какой-то из ответов true - возвратить true

// если дошли до конца цепочки - false

В примере выше rabbit.\_\_proto\_\_ === Rabbit.prototype, так что результат будет получен немедленно. В случае с наследованием, совпадение будет на втором шаге:

class Animal {}

class Rabbit extends Animal {}

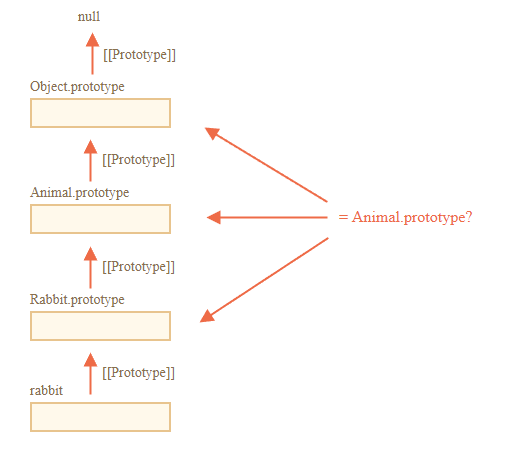
let rabbit = new Rabbit();

alert(rabbit instanceof Animal); // true

// rabbit.\_\_proto\_\_ === Rabbit.prototype

// rabbit.\_\_proto\_\_.\_\_proto\_\_ === Animal.prototype

Вот иллюстрация того как rabbit instanceof Animal сравнивается с Animal.prototype:



Есть метод [objA.isPrototypeOf(objB)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/object/isPrototypeOf), который возвращает true, если объект objA есть где-то в прототипной цепочке объекта objB. Так что obj instanceof Class можно перефразировать как Class.prototype.isPrototypeOf(obj). Сам конструктор Class не участвует в процессе проверки. Важна только цепочка прототипов Class.prototype. Это может приводить к интересным последствиям при изменении свойства prototype после создания объекта. Как, например, тут:

function Rabbit() {}

let rabbit = new Rabbit();

// заменяем прототип

Rabbit.prototype = {};

alert( rabbit instanceof Rabbit ); // false

Известно, что обычные объекты преобразуется к строке как [object Object]:

let obj = {};

alert(obj); // [object Object]

alert(obj.toString()); // то же самое

Так работает реализация метода toString. Но у toString имеются скрытые возможности, которые делают метод гораздо более мощным. Можно использовать его как расширенную версию typeof и как альтернативу instanceof. Согласно [спецификации](https://tc39.github.io/ecma262/#sec-object.prototype.tostring) встроенный метод toString может бы позаимствован у объекта и вызван в контексте любого другого значения. И результат зависит от типа этого значения.

* для числа это будет [object Number];
* для булева типа это будет [object Boolean];
* для null: [object Null];
* для undefined: [object Undefined];
* для массивов: [object Array] и т.д.

Например:

let objectToString = Object.prototype.toString;

let arr = [];

alert( objectToString.call(arr) ); // [object Array]

В примере использовался [call](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/function/call), чтобы выполнить функцию objectToString в контексте this=arr. Внутри, алгоритм метода toString анализирует контекст вызова this и возвращает соответствующий результат. Пример:

let s = Object.prototype.toString;

alert( s.call(123) ); // [object Number]

alert( s.call(null) ); // [object Null]

alert( s.call(alert) ); // [object Function]

**Свойство** [**Symbol.toStringTag**](https://learn.javascript.ru/instanceof#symbol-tostringtag)

Поведение метода объектов toString можно настраивать, используя специальное свойство объекта Symbol.toStringTag. Например:

let user = {

[Symbol.toStringTag]: "User"

};

alert( {}.toString.call(user) ); // [object User]

Такое свойство есть у большей части объектов, специфичных для определённых окружений. Вот несколько примеров для браузера:

alert( window[Symbol.toStringTag]); // window

alert( XMLHttpRequest.prototype[Symbol.toStringTag] ); // XMLHttpRequest

alert( {}.toString.call(window) ); // [object Window]

alert( {}.toString.call(new XMLHttpRequest()) ); // [object XMLHttpRequest]

Как видно, результат – это значение Symbol.toStringTag (если он имеется) обёрнутое в [object ...]. В итоге получили typeof, который не только работает с примитивными типами данных, но также и со встроенными объектами, и даже может быть настроен.

Можно использовать {}.toString.call вместо instanceof для встроенных объектов, когда надо получить тип в виде строки, а не просто сделать проверку.

1. [**Примеси**](https://learn.javascript.ru/mixins)

В JavaScript можно наследовать только от одного объекта. Объект имеет единственный [[Prototype]]. И класс может расширить только один другой класс. Иногда это ограничивает разработчика. Например, есть класс User, который реализует пользователей, и класс EventEmitter, реализующий события. Надо добавить функционал класса EventEmitter к User, чтобы пользователи могли легко генерировать события.

Для таких случаев существуют «примеси». По определению из Википедии, [примесь](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BC%D0%B5%D1%81%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) – это класс, методы которого предназначены для использования в других классах, причём без наследования от примеси. Другими словами, примесь определяет методы, которые реализуют определённое поведение. Примесь не используется сама по себе, а используется, чтобы добавить функционал другим классам.

Простейший способ реализовать примесь в JavaScript – это создать объект с полезными методами, которые затем могут быть легко добавлены в прототип любого класса. В примере ниже примесь sayHiMixin имеет методы для придания объектам класса User возможности вести разговор:

let sayHiMixin = {

sayHi() {

alert(`Привет, ${this.name}`);

},

sayBye() {

alert(`Пока, ${this.name}`);

}

};

// использование:

class User {

constructor(name) {

this.name = name;

}

}

// копируем методы

Object.assign(User.prototype, sayHiMixin);

new User("Вася").sayHi(); // Привет, Вася!

Это не наследование, а просто копирование методов. Таким образом, класс User может наследовать от другого класса, но при этом также включать в себя примеси, «подмешивающие» другие методы, например:

class User extends Person {

// ...

}

Object.assign(User.prototype, sayHiMixin);

Примеси могут наследовать друг друга. В примере ниже sayHiMixin наследует от sayMixin:

let sayMixin = {

say(phrase) {

alert(phrase);

}

};

let sayHiMixin = {

\_\_proto\_\_: sayMixin,

sayHi() {

// вызываем метод родителя

super.say(`Привет, ${this.name}`); // (\*)

},

sayBye() {

super.say(`Пока, ${this.name}`); // (\*)

}

};

class User {

constructor(name) {

this.name = name;

}

}

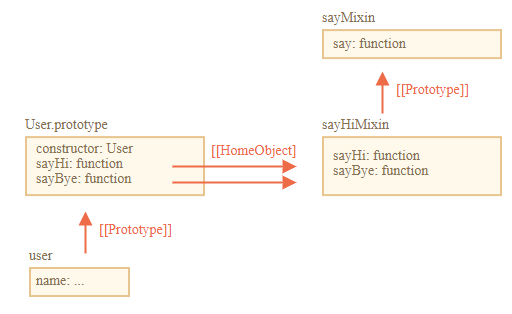
// копируем методы

Object.assign(User.prototype, sayHiMixin);

new User("Вася").sayHi(); // Привет, Вася!

Обратите внимание, что при вызове родительского метода super.say() из sayHiMixin (строки, помеченные (\*)) этот метод ищется в прототипе самой примеси, а не класса.

Диаграмма:



Это связано с тем, что методы sayHi и sayBye были изначально созданы в объекте sayHiMixin. Несмотря на то, что они скопированы, их внутреннее свойство [[HomeObject]] ссылается на sayHiMixin, как показано на картинке выше. Так как super ищет родительские методы в [[HomeObject]].[[Prototype]], это означает sayHiMixin.[[Prototype]], а не User.[[Prototype]].

[**EventMixin**](https://learn.javascript.ru/mixins#eventmixin)

Многие объекты в браузерной разработке (и не только) обладают важной способностью – они могут генерировать события. События – отличный способ передачи информации всем, кто в ней заинтересован. Создадим примесь, которая позволит легко добавлять функционал по работе с событиями любым классам/объектам.

* Примесь добавит метод .trigger(name, [data]) для генерации события. Аргумент name – это имя события, за которым могут следовать другие аргументы с данными для события.
* Также будет добавлен метод .on(name, handler), который назначает обработчик для события с заданным именем. Обработчик будет вызван, когда произойдёт событие с указанным именем name, и получит данные из .trigger.
* Будет добавлен метод .off(name, handler), который удаляет обработчик указанного события.

После того, как все методы примеси будут добавлены, объект user сможет сгенерировать событие "login" после входа пользователя в личный кабинет. А другой объект, к примеру, calendar сможет использовать это событие, чтобы показывать зашедшему пользователю актуальный для него календарь. Или menu может генерировать событие "select", когда элемент меню выбран, а другие объекты могут назначать обработчики, чтобы реагировать на это событие, и т.п. Код примеси:

let eventMixin = {

/\*\*

\* Подписаться на событие, использование:

\* menu.on('select', function(item) { ... }

\*/

on(eventName, handler) {

if (!this.\_eventHandlers) this.\_eventHandlers = {};

if (!this.\_eventHandlers[eventName]) {

this.\_eventHandlers[eventName] = [];

}

this.\_eventHandlers[eventName].push(handler);

},

/\*\*

\* Отменить подписку, использование:

\* menu.off('select', handler)

\*/

off(eventName, handler) {

let handlers = this.\_eventHandlers && this.\_eventHandlers[eventName];

if (!handlers) return;

for (let i = 0; i < handlers.length; i++) {

if (handlers[i] === handler) {

handlers.splice(i--, 1);

}

}

},

/\*\*

\* Сгенерировать событие с указанным именем и данными

\* this.trigger('select', data1, data2);

\*/

trigger(eventName, ...args) {

if (!this.\_eventHandlers || !this.\_eventHandlers[eventName]) {

return; // обработчиков для этого события нет

}

// вызов обработчиков

this.\_eventHandlers[eventName].forEach(handler => handler.apply(this, args));

}

};

Использование:

class Menu {

choose(value) {

this.trigger("select", value);

}

}

Object.assign(Menu.prototype, eventMixin);

let menu = new Menu();

// Добавить обработчик, который будет вызван при событии "select":

menu.on("select", value => alert(`Выбранное значение: ${value}`));

// Генерирует событие => обработчик выше запускается и выводит:

menu.choose("123"); // Выбранное значение: 123

Теперь если есть код, заинтересованный в событии "select", то он может слушать его с помощью menu.on(...). А eventMixin позволяет легко добавить такое поведение в любой класс без вмешательства в цепочку наследования.

1. **Обработка ошибок, "try..catch"**

В JavaScript есть синтаксическая конструкция try..catch, которая позволяет обрабатывать ошибки, в случае их возникновения.

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

Конструкция try..catch состоит из двух основных блоков: try, и затем catch:

try {

// код...

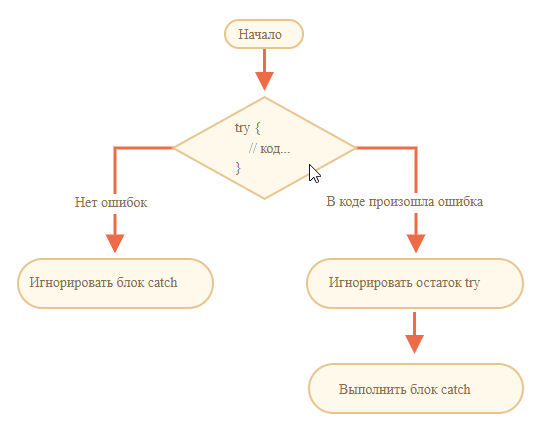
} catch (err) {

// обработка ошибки

}

Работает она так:

1. Сначала выполняется код внутри блока try {...}.
2. Если в нём нет ошибок, то блок catch(err) игнорируется: выполнение доходит до конца try и потом далее, полностью пропуская catch.
3. Если же в нём возникает ошибка, то выполнение try прерывается, и поток управления переходит в начало catch(err). Переменная err (можно использовать любое имя) содержит объект ошибки с подробной информацией о произошедшем.



Таким образом, при ошибке в блоке try {…} скрипт не «падает», и есть возможность обработать ошибку внутри catch. Рассмотрим примеры.

Пример без ошибок: выведет alert (1) и (2):

try {

alert('Начало блока try'); // (1) <--

// код без ошибок

alert('Конец блока try'); // (2) <--

} catch(err) {

alert('Catch игнорируется, так как нет ошибок'); // (3)

}

Пример с ошибками: выведет (1) и (3):

try {

alert('Начало блока try'); // (1) <--

lalala; // ошибка, переменная не определена

alert('Конец блока try (никогда не выполнится)'); // (2)

} catch(err) {

alert(`Возникла ошибка!`); // (3) <--

}

Чтобы try..catch работал, код должен быть выполнимым. Другими словами, это должен быть корректный JavaScript-код. Он не сработает, если код синтаксически неверен, например, содержит несовпадающее количество фигурных скобок:

try {

{{{{{{{{{{{{

} catch(e) {

alert("Движок не может понять этот код, он не корректен");

}

JavaScript-движок сначала читает код, а затем исполняет его. Ошибки, которые возникают во время фазы чтения, называются ошибками парсинга. Их нельзя обработать (изнутри этого кода), потому что движок не понимает код. Таким образом, try..catch может обрабатывать только ошибки, которые возникают в корректном коде. Такие ошибки называют «ошибками во время выполнения», а иногда «исключениями».

Исключение, которое произойдёт в коде, запланированном «на будущее», например в setTimeout, try..catch не поймает:

try {

setTimeout(function() {

noSuchVariable; // скрипт «упадёт» тут

}, 1000);

} catch (e) {

alert( "не сработает" );

}

Это потому, что функция выполняется позже, когда движок уже покинул конструкцию try..catch. Чтобы поймать исключение внутри запланированной функции, try..catch должен находиться внутри самой этой функции:

setTimeout(function() {

try {

noSuchVariable; // try..catch обрабатывает ошибку

} catch {

alert( "ошибка поймана!" );

}

}, 1000);

[**Объект ошибки**](https://learn.javascript.ru/try-catch#obekt-oshibki)

Когда возникает ошибка, JavaScript генерирует объект, содержащий её детали. Затем этот объект передаётся как аргумент в блок catch:

try {

// ...

} catch(err) { // объект ошибки, можно использовать другое название вместо err

// ...

}

Для всех встроенных ошибок этот объект имеет два основных свойства:

* name – имя ошибки. Например, для неопределённой переменной это "ReferenceError".
* message – текстовое сообщение о деталях ошибки.

В большинстве окружений доступны и другие, нестандартные свойства. Одно из самых широко используемых и поддерживаемых – это stack – текущий стек вызова: строка, содержащая информацию о последовательности вложенных вызовов, которые привели к ошибке. Используется в целях отладки.

Например:

try {

lalala; // ошибка, переменная не определена

} catch(err) {

alert(err.name); // ReferenceError

alert(err.message); // lalala is not defined

alert(err.stack); // ReferenceError: lalala is not defined at (...стек вызовов)

// Ошибка приводится к строке вида "name: message"

alert(err); // ReferenceError: lalala is not defined

}

[**Блок «catch» без переменной**](https://learn.javascript.ru/try-catch#blok-catch-bez-peremennoy)

Эта возможность была добавлена в язык недавно. В старых браузерах может понадобиться полифилл. Если не нужны детали ошибки, в catch можно её пропустить:

try {

// ...

} catch { // без (err)

// ...

}

Рассмотрим реальные случаи использования try..catch. Как известно, JavaScript поддерживает метод [JSON.parse(str)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse) для чтения JSON. Обычно он используется для декодирования данных, полученных по сети, от сервера или из другого источника:

let json = '{"name":"John", "age": 30}';

let user = JSON.parse(json);

alert( user.name ); // John

alert( user.age ); // 30

Если json некорректен, JSON.parse генерирует ошибку, то есть скрипт «падает». Это значит, что если вдруг что-то не так с данными, то посетитель никогда (если, конечно, не откроет консоль) об этом не узнает. Используем try..catch для обработки ошибки:

let json = "{ некорректный JSON }";

try {

let user = JSON.parse(json); // ошибка

alert( user.name ); // не сработает

} catch (e) {

// выполнение «прыгает» сюда

alert( "Извините, в данных ошибка, мы попробуем получить их ещё раз." );

alert( e.name );

alert( e.message );

}

Здесь использовался блок catch только для вывода сообщения, но также можно сделать гораздо больше: отправить новый сетевой запрос, предложить посетителю альтернативный способ, отослать информацию об ошибке на сервер для логирования,

[**Генерация собственных ошибок**](https://learn.javascript.ru/try-catch#generatsiya-sobstvennyh-oshibok)

Возможна ситуация, когда json синтаксически корректен, но не содержит необходимого свойства name. Например, так:

let json = '{ "age": 30 }';

try {

let user = JSON.parse(json);

alert( user.name ); // нет свойства name

} catch (e) {

alert( "не выполнится" );

}

Здесь JSON.parse выполнится без ошибок, но на самом деле отсутствие свойства name это ошибка. Для того, чтобы унифицировать обработку ошибок, необходимо воспользоваться оператором throw.

[**Оператор «throw»**](https://learn.javascript.ru/try-catch#operator-throw)

Оператор throw генерирует ошибку.

Синтаксис:

throw <объект ошибки>

Технически в качестве объекта ошибки можно передать что угодно. Это может быть даже примитив, число или строка, но всё же лучше, чтобы это был объект, желательно со свойствами name и message (для совместимости со встроенными ошибками).

В JavaScript есть множество встроенных конструкторов для стандартных ошибок: Error, SyntaxError, ReferenceError, TypeError и другие. Можно использовать и их для создания объектов ошибки. Их синтаксис:

let error = new Error(message);

// или

let error = new SyntaxError(message);

let error = new ReferenceError(message);

// ...

Для встроенных ошибок (не для любых объектов, только для ошибок), свойство name – это в точности имя конструктора. А свойство message берётся из аргумента. Например:

let error = new Error(" Ого, ошибка! o\_O");

alert(error.name); // Error

alert(error.message); // Ого, ошибка! o\_O

Посмотрим, какую ошибку генерирует JSON.parse:

try {

JSON.parse("{ bad json o\_O }");

} catch(e) {

alert(e.name); // SyntaxError

alert(e.message); // Unexpected token o in JSON at position 0

}

Как видно, это SyntaxError. В случае отсутствие свойства name – это ошибка, ведь пользователи должны иметь имена. Сгенерируем её:

let json = '{ "age": 30 }';

try {

let user = JSON.parse(json);

if (!user.name) {

throw new SyntaxError("Данные неполны: нет имени"); // (\*)

}

alert( user.name );

} catch(e) {

alert( "JSON Error: " + e.message ); // JSON Error: Данные неполны: нет имени

}

В строке (\*) оператор throw генерирует ошибку SyntaxError с сообщением message. Точно такого же вида, как генерирует сам JavaScript. Выполнение блока try немедленно останавливается, и поток управления прыгает в catch. Теперь блок catch становится единственным местом для обработки всех ошибок: и для JSON.parse и для других случаев.

**[Проброс исключения](https://learn.javascript.ru/try-catch" \l "probros-isklyucheniya)**

В примере выше использовалась конструкция try..catch для обработки некорректных данных. Предположим, что в блоке try {...}возникнет другая неожиданная ошибка, например, программная (неопределённая переменная) или какая-то ещё, а не ошибка, связанная с некорректными данными. Пример:

let json = '{ "age": 30 }';

try {

user = JSON.parse(json); // забыл добавить "let" перед user

// ...

} catch(err) {

alert("JSON Error: " + err); // JSON Error: ReferenceError: user is not defined

// (не JSON ошибка на самом деле)

}

В рассматриваемом примере catch получает из try ошибку, не связанную с данными, но показывает то же самое сообщение "JSON Error". Это неправильно и затрудняет отладку кода. Есть способ выяснить, какая ошибка была получена, например, по её свойству name:

try {

user = { /\*...\*/ };

} catch(e) {

alert(e.name); // "ReferenceError" из-за неопределённой переменной

}

Есть простое правило: блок catch должен обрабатывать только те ошибки, которые ему известны, и «пробрасывать» все остальные.

Техника «проброс исключения» выглядит так:

1. Блок catch получает все ошибки.
2. В блоке catch(err) {...} анализируется объект ошибки err.
3. Если неизвестно как её обработать, тогда генерируется throw err.

В коде ниже используется проброс исключения, catch обрабатывает только SyntaxError:

let json = '{ "age": 30 }';

try {

let user = JSON.parse(json);

if (!user.name) {

throw new SyntaxError("Данные неполны: нет имени");

}

blabla(); // неожиданная ошибка

alert( user.name );

} catch(e) {

if (e.name == "SyntaxError") {

alert( "JSON Error: " + e.message );

} else {

throw e; // rethrow (\*)

}

}

Ошибка в строке (\*) из блока catch «выпадает наружу» и может быть поймана другой внешней конструкцией try..catch (если есть), или «убьёт» скрипт. Таким образом, блок catch фактически обрабатывает только те ошибки, с которыми он знает, как справляться, и пропускает остальные. Пример ниже демонстрирует, как такие ошибки могут быть пойманы с помощью ещё одного уровня try..catch:

function readData() {

let json = '{ "age": 30 }';

try {

// ...

blabla(); // ошибка

} catch (e) {

// ...

if (e.name != 'SyntaxError') {

throw e; // проброс исключения

}

}

}

try {

readData();

} catch (e) {

alert( "Внешний catch поймал: " + e ); // поймал

}

В рассматриваемом примере в readData обрабатываются только SyntaxError, тогда как внешний блок try..catch обрабатывает всё.

**Конструкция** [**try…catch…finally**](https://learn.javascript.ru/try-catch#try-catch-finally)

Конструкция try..catch может содержать ещё одну секцию: finally. Если секция есть, то она выполняется в любом случае:

* после try, если не было ошибок,
* после catch, если ошибки были.

Расширенный синтаксис выглядит следующим образом:

try {

//... пробуем выполнить код...

} catch(e) {

//... обрабатываем ошибки ...

} finally {

//... выполняем всегда ...

}

Рассмотрим следующий код:

try {

alert( 'try' );

if (confirm('Сгенерировать ошибку?')) BAD\_CODE();

} catch (e) {

alert( 'catch' );

} finally {

alert( 'finally' );

}

У рассматриваемого кода есть два пути выполнения:

1. Если ответить на вопрос «Сгенерировать ошибку?» утвердительно, то try -> catch -> finally.
2. Если ответить отрицательно, то try -> finally.

Секцию finally часто используют, если надо что-то выполнить вне зависимости от того, будет ошибка или нет. Например, надо измерить время, которое занимает функция чисел Фибоначчи fib(n). Естественно, можно начать измерения до того, как функция начнёт выполняться и закончить после. Но если при вызове функции возникла ошибка, (в частности, реализация fib(n) в коде ниже возвращает ошибку для отрицательных и для нецелых чисел), то следует использовать секцию finally, которая завершит измерения несмотря ни на что.

Здесь finally гарантирует, что время будет измерено корректно в обеих ситуациях – и в случае успешного завершения fib и в случае ошибки:

let num = +prompt("Введите положительное целое число?", 35)

let diff, result;

function fib(n) {

if (n < 0 || Math.trunc(n) != n) {

throw new Error("Должно быть целое неотрицательное число");

}

return n <= 1 ? n : fib(n - 1) + fib(n - 2);

}

let start = Date.now();

try {

result = fib(num);

} catch (e) {

result = 0;

} finally {

diff = Date.now() - start;

}

alert(result || "возникла ошибка");

alert( `Выполнение заняло ${diff}ms` );

Это можно проверить, запустив этот код и введя 35 в prompt – код завершится нормально, finally выполнится после try. Затем ввести -1 – незамедлительно произойдёт ошибка, выполнение займёт 0ms. Оба измерения выполняются корректно. Другими словами, неважно как завершилась функция: через return или throw. Секция finally срабатывает в обоих случаях.

Переменные внутри try..catch..finally локальны. Обратите внимание, что переменные result и diff в коде выше объявлены до try..catch. Если переменную объявить в блоке, например, в try, то она не будет доступна после него.

Блок finally срабатывает при любом выходе из try..catch, в том числе и return. В примере ниже из try происходит return, но finally получает управление до того, как контроль возвращается во внешний код.

function func() {

try {

return 1;

} catch (e) {

/\* ... \*/

} finally {

alert( 'finally' );

}

}

alert( func() ); // сначала срабатывает alert из finally, а затем этот код

Конструкция try..finally без секции catch также полезна. Стоит применять её, когда нет необходимости обрабатывать ошибки (пусть выпадут), но надо убедиться, что начатые процессы завершились.

function func() {

// начать делать что-то, что требует завершения

try {

// ...

} finally {

// завершить это, даже если все упадёт

}

}

В приведённом выше коде ошибка всегда выпадает наружу, потому что тут нет блока catch. Но finally отрабатывает до того, как поток управления выйдет из функции.

1. **Пользовательские ошибки, расширение Error**

Разработчикам часто необходимы собственные классы ошибок для разных задач. Для ошибок при работе с сетью может понадобиться HttpError, для операций с базой данных DbError, для поиска – NotFoundError и т.д. Ошибки должны поддерживать базовые свойства, такие как message, name и, желательно, stack. Но также они могут иметь свои собственные свойства. Например, объекты HttpError могут иметь свойство statusCode со значениями 404, 403 или 500.

JavaScript позволяет вызывать throw с любыми аргументами, то есть технически пользовательские классы ошибок не нуждаются в наследовании от Error. Но если использовать наследование, то появляется возможность идентификации объектов ошибок посредством obj instanceof Error. Так что лучше применять наследование.

По мере роста приложения, наши собственные ошибки образуют иерархию, например, HttpTimeoutError может наследовать от HttpError и так далее.

[**Расширение Error**](https://learn.javascript.ru/custom-errors#rasshirenie-error)

В качестве примера рассмотрим функцию readUser(json), которая должна читать данные пользователя в формате JSON. Пример того, как может выглядеть корректный json:

let json = `{ "name": "John", "age": 30 }`;

Внутри используется JSON.parse. При получении некорректного json он будет генерировать ошибку SyntaxError. Но даже если json синтаксически верен, то это не значит, что это будет корректный пользователь. Могут быть пропущены необходимые данные. Например, могут отсутствовать свойства name и age, которые являются необходимыми для наших пользователей. Функция readUser(json) будет не только читать JSON-данные, но и проверять их («валидировать»). Если необходимые поля отсутствуют или данные в неверном формате, то это будет ошибкой. Но не синтаксической ошибкой SyntaxError, потому что данные синтаксически корректны. Это будет другая ошибка.

Назовем её ошибкой валидации ValidationError и создадим для неё класс. Ошибка этого вида должна содержать информацию о поле, которое является источником ошибки. Класс ValidationError должен наследовать от встроенного класса Error. Класс Error встроенный, вот его примерный код, чтобы понимать, что расширяется:

class Error {

constructor(message) {

this.message = message;

this.name = "Error";

this.stack = <стек вызовов>;

}

}

Теперь унаследуем от него ValidationError и попробуем новый класс в действии:

class ValidationError extends Error {

constructor(message) {

super(message); // (1)

this.name = "ValidationError"; // (2)

}

}

function test() {

throw new ValidationError("Упс!");

}

try {

test();

} catch(err) {

alert(err.message);

alert(err.name); // ValidationError

alert(err.stack); // список вложенных вызовов с номерами строк для каждого

}

Обратите внимание: в строке (1) вызывается родительский конструктор. JavaScript требует вызов super в дочернем конструкторе, так что это обязательно. Родительский конструктор устанавливает свойство message. Родительский конструктор также устанавливает свойство name для "Error", поэтому в строке (2) оно сбрасывается на правильное значение. Попробуем использовать его в readUser(json):

class ValidationError extends Error {

constructor(message) {

super(message);

this.name = "ValidationError";

}

}

// Использование

function readUser(json) {

let user = JSON.parse(json);

if (!user.age) {

throw new ValidationError("Нет поля: age");

}

if (!user.name) {

throw new ValidationError("Нет поля: name");

}

return user;

}

// Рабочий пример с try..catch

try {

let user = readUser('{ "age": 25 }');

} catch (err) {

if (err instanceof ValidationError) {

alert("Некорректные данные: " + err.message); // Некорректные данные: Нет поля: name

} else if (err instanceof SyntaxError) { // (\*)

alert("JSON Ошибка Синтаксиса: " + err.message);

} else {

throw err; // неизвестная ошибка, пробросить исключение (\*\*)

}

}

Блок try..catch в коде выше обрабатывает и нашу ValidationError, и встроенную SyntaxError из JSON.parse. Обратите внимание, как используется instanceof для проверки конкретного типа ошибки в строке (\*). Можно также проверить тип, используя err.name:

// ...

// вместо (err instanceof SyntaxError)

} else if (err.name == "SyntaxError") { // (\*)

// ...

Версия с instanceof гораздо лучше, потому что в будущем можно расширить ValidationError, сделав его подтипы, такие как PropertyRequiredError. И проверка instanceof продолжит работать для новых наследованных классов. Также важно, что если catch встречает неизвестную ошибку, то он пробрасывает её в строке (\*\*). Блок catch знает, только как обрабатывать ошибки валидации и синтаксические ошибки, а другие виды ошибок (из-за опечаток в коде и др.) он должен пропустить.

Класс ValidationError является общим и не учитывает ряд ошибок. Например, отсутствие свойства или неверный формат (например, строка как значение возраста age). Поэтому для отсутствующих свойств сделаем более конкретный класс PropertyRequiredError. Он будет нести дополнительную информацию о свойстве, которое отсутствует.

class ValidationError extends Error {

constructor(message) {

super(message);

this.name = "ValidationError";

}

}

class PropertyRequiredError extends ValidationError {

constructor(property) {

super("Нет свойства: " + property);

this.name = "PropertyRequiredError";

this.property = property;

}

}

// Применение

function readUser(json) {

let user = JSON.parse(json);

if (!user.age) {

throw new PropertyRequiredError("age");

}

if (!user.name) {

throw new PropertyRequiredError("name");

}

return user;

}

// Рабочий пример с try..catch

try {

let user = readUser('{ "age": 25 }');

} catch (err) {

if (err instanceof ValidationError) {

alert("Неверные данные: " + err.message); // Неверные данные: Нет свойства: name

alert(err.name); // PropertyRequiredError

alert(err.property); // name

} else if (err instanceof SyntaxError) {

alert("Ошибка синтаксиса JSON: " + err.message);

} else {

throw err; // неизвестная ошибка, повторно выбросит исключение

}

}

Новый класс PropertyRequiredError очень просто использовать: необходимо указать только имя свойства new PropertyRequiredError(property). Сообщение для пользователя message генерируется конструктором. Обратите внимание, что свойство this.name в конструкторе PropertyRequiredError снова присвоено вручную. Правда, немного утомительно – присваивать this.name = <class name> в каждом классе пользовательской ошибки. Можно этого избежать, если сделать собственный «базовый» класс ошибки MyError, который будет ставить this.name = this.constructor.name. И затем наследовать все ошибки уже от него. Вот упрощённый код с MyError и другими пользовательскими классами ошибок:

class MyError extends Error {

constructor(message) {

super(message);

this.name = this.constructor.name;

}

}

class ValidationError extends MyError { }

class PropertyRequiredError extends ValidationError {

constructor(property) {

super("Нет свойства: " + property);

this.property = property;

}

}

// name корректное

alert( new PropertyRequiredError("field").name ); // PropertyRequiredError

Теперь пользовательские ошибки стали намного короче, особенно ValidationError, так как нет строки "this.name = ..." в конструкторе.

[**Обёртывание исключений**](https://learn.javascript.ru/custom-errors#obyortyvanie-isklyucheniy)

Назначение функции readUser в приведенном выше коде – это «чтение данных пользователя». В процессе могут возникнуть различные виды ошибок. Сейчас есть SyntaxError и ValidationError, но в будущем функция readUser может расшириться и, возможно, генерировать другие виды ошибок. Код, который вызывает readUser, должен обрабатывать эти ошибки.

Сейчас в нём используются проверки if в блоке catch, которые проверяют класс и обрабатывают известные ошибки и пробрасывают дальше неизвестные. Но если функция readUser генерирует несколько видов ошибок, не стоит проверять все типы ошибок поодиночке во всех местах в коде, где вызывается readUser. Лучше обработать какую-то обобщённую ошибку чтения данных.

Создадим новый класс ReadError для представления таких ошибок. Если ошибка возникает внутри readUser, её надо перехватить и сгенерировать ReadError. Также ссылка на исходную ошибку будет сохранена в свойстве cause. Тогда внешний код должен будет только проверить наличие ReadError. Этот код определяет ошибку ReadError и демонстрирует её использование в readUserи try..catch:

class ReadError extends Error {

constructor(message, cause) {

super(message);

this.cause = cause;

this.name = 'ReadError';

}

}

class ValidationError extends Error { /\*...\*/ }

class PropertyRequiredError extends ValidationError { /\* ... \*/ }

function validateUser(user) {

if (!user.age) {

throw new PropertyRequiredError("age");

}

if (!user.name) {

throw new PropertyRequiredError("name");

}

}

function readUser(json) {

let user;

try {

user = JSON.parse(json);

} catch (err) {

if (err instanceof SyntaxError) {

throw new ReadError("Синтаксическая ошибка", err);

} else {

throw err;

}

}

try {

validateUser(user);

} catch (err) {

if (err instanceof ValidationError) {

throw new ReadError("Ошибка валидация", err);

} else {

throw err;

}

}

}

try {

readUser('{bad json}');

} catch (e) {

if (e instanceof ReadError) {

alert(e);

// Исходная ошибка: SyntaxError:Unexpected token b in JSON at position 1

alert("Исходная ошибка: " + e.cause);

} else {

throw e;

}

}

В приведённом выше коде readUser работает так, как описано – функция распознаёт синтаксические ошибки и ошибки валидации и выдаёт вместо них ошибки ReadError (неизвестные ошибки, как обычно, пробрасываются). Внешний код проверяет только instanceof ReadError. Не нужно перечислять все возможные типы ошибок

Этот подход называется «обёртывание исключений», потому что «исключения низкого уровня» «оборачиваются» в ReadError, который является более абстрактным и более удобным для использования в вызывающем коде. Такой подход широко используется в объектно-ориентированном программировании.

1. **Модули: введение.**

По мере роста приложения, обычно возникает необходимость разделить его на много файлов, так называемых «модулей». Модуль обычно содержит класс или библиотеку с функциями. Долгое время в JavaScript отсутствовал синтаксис модулей на уровне языка. Это не было проблемой, потому что первые скрипты были маленькими и простыми. В модулях не было необходимости. Но со временем скрипты становились всё более и более сложными, поэтому сообщество придумало несколько вариантов организации кода в модули. Появились библиотеки для динамической подгрузки модулей. Например:

* [AMD](https://ru.wikipedia.org/wiki/Asynchronous_module_definition) – одна из самых старых модульных систем, изначально реализована библиотекой [require.js](http://requirejs.org/).
* [CommonJS](http://wiki.commonjs.org/wiki/Modules/1.1) – модульная система, созданная для сервера Node.js.
* [UMD](https://github.com/umdjs/umd) – ещё одна модульная система, предлагается как универсальная, совместима с AMD и CommonJS.

Теперь все они постепенно становятся частью истории, хотя их и можно найти в старых скриптах.

Система модулей на уровне языка появилась в стандарте JavaScript в 2015 году и постепенно эволюционировала. На данный момент она поддерживается большинством браузеров и Node.js.

Модуль – это файл с кодом. Один скрипт – это один модуль. Модули могут загружать друг друга и использовать директивы export и import, чтобы обмениваться функциональностью, вызывать функции одного модуля из другого:

* export отмечает переменные и функции, которые должны быть доступны вне текущего модуля.
* import позволяет импортировать функциональность из других модулей.

Например, если есть файл sayHi.js, который экспортирует функцию:

//sayHi.js

export function sayHi(user) {

alert(`Hello, ${user}!`);

}

Тогда другой файл может импортировать её и использовать:

// main.js

import {sayHi} from './sayHi.js';

alert(sayHi); // function...

sayHi('John'); // Hello, John!

Директива import загружает модуль по пути ./sayHi.js относительно текущего файла и записывает эксортированную функцию sayHi в соответствующую переменную. Так как модули поддерживают ряд специальных ключевых слов, и у них есть ряд особенностей, то необходимо явно сказать браузеру, что скрипт является модулем, при помощи атрибута <script type="module">. Вот так:

// say.js

export function sayHi(user) {

return `Hello, ${user}!`;

}

// index.html

<!doctype html>

<script type="module">

import {sayHi} from './say.js';

document.body.innerHTML = sayHi('John');

</script>

Браузер автоматически загрузит и запустит импортированный модуль (и те, которые он импортирует, если надо), а затем запустит скрипт.



[**Основные возможности модулей**](https://learn.javascript.ru/modules-intro#osnovnye-vozmozhnosti-moduley)

Есть основные возможности и особенности, работающие как в браузере, так и в серверном JavaScript.

В модулях всегда используется режим use strict. Например, присваивание к необъявленной переменной вызовет ошибку.

<script type="module">

a = 5; // ошибка

</script>

Каждый модуль имеет свою собственную область видимости. Другими словами, переменные и функции, объявленные в модуле, не видны в других скриптах.

В следующем примере импортированы 2 скрипта, и hello.js пытается использовать переменную user, объявленную в user.js. В итоге ошибка:



// hello.js

alert(user);

// user.js

let user = "John";

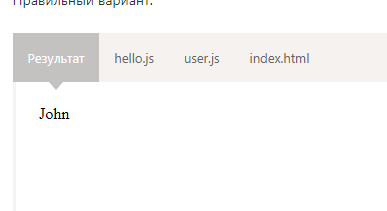
// index.html

<!doctype html>

<script type="module" src="user.js"></script>

<script type="module" src="hello.js"></script>

Модули должны экспортировать функционал, предназначенный для использования извне. А другие модули могут его импортировать. Так что надо импортировать user.js в hello.js и взять из него нужный функционал, вместо того чтобы полагаться на глобальные переменные. Правильный вариант:



// hello.js

import {user} from './user.js';

document.body.innerHTML = user; // John

// user.js

export let user = "John";

// index.html

import {user} from './user.js';

document.body.innerHTML = user; // John

В браузере также существует независимая область видимости для каждого скрипта <script type="module">:

<script type="module">

// Переменная доступна только в этом модуле

let user = "John";

</script>

<script type="module">

alert(user); // Error: user is not defined

</script>

Если нужно сделать глобальную переменную уровня всей страницы, можно явно присвоить её объекту window, тогда получить значение переменной можно обратившись к window.user. Но это должно быть исключением, требующим веской причины.

Если один и тот же модуль используется в нескольких местах, то его код выполнится только один раз, после чего экспортируемая функциональность передаётся всем импортёрам. Это очень важно для понимания работы модулей. Рассмотрим примеры.

Во-первых, если при запуске модуля возникают побочные эффекты, например, выдаётся сообщение, то импорт модуля в нескольких местах покажет его только один раз – при первом импорте:

// alert.js

alert("Модуль выполнен!");

// Импорт одного и того же модуля в разных файлах

// 1.js

import `./alert.js`; // Модуль выполнен!

// 2.js

import `./alert.js`; // (ничего не покажет)

На практике, задача кода модуля – это обычно инициализация, создание внутренних структур данных, а если надо, чтобы что-то можно было использовать много раз, то экспортируем это.

Рассмотрим более сложный пример. Представим, что модуль экспортирует объект:

// admin.js

export let admin = {

name: "John"

};

Если модуль импортируется в нескольких файлах, то код модуля будет выполнен только один раз, объект admin будет создан и в дальнейшем будет передан всем импортёрам. Все импортёры получат один-единственный объект admin:

// 1.js

import {admin} from './admin.js';

admin.name = "Pete";

// 2.js

import {admin} from './admin.js';

alert(admin.name); // Pete

В примере оба файла, 1.js и 2.js, импортируют один и тот же объект. Изменения, сделанные в 1.js, будут видны в 2.js. Если что-то изменится в объекте admin, то другие модули тоже увидят эти изменения. Такое поведение позволяет конфигурировать модули при первом импорте. Можно установить его свойства один раз, и в дальнейших импортах он будет уже настроенным.

Например, модуль admin.js предоставляет определённую функциональность, но ожидает передачи учётных данных в объект admin извне:

// admin.js

export let admin = { };

export function sayHi() {

alert(`Ready to serve, ${admin.name}!`);

}

В init.js, первом скрипте рассматриваемого приложения, установим admin.name. Тогда все это увидят, включая вызовы, сделанные из самого admin.js:

// init.js

import {admin} from './admin.js';

admin.name = "Pete";

Другой модуль тоже увидит admin.name:

// other.js

import {admin, sayHi} from './admin.js';

alert(admin.name); // Pete

sayHi(); // Ready to serve, Pete!

Объект import.meta содержит информацию о текущем модуле. Содержимое зависит от окружения. В браузере он содержит ссылку на скрипт или ссылку на текущую веб-страницу, если модуль встроен в HTML:

<script type="module">

alert(import.meta.url); // ссылка на html страницу для встроенного скрипта

</script>

В модуле на верхнем уровне this не определён (undefined). Это незначительная особенность, но для полноты картины нужно упомянуть об этом. Сравним с немодульными скриптами, там this – глобальный объект:

<script>

alert(this); // window

</script>

<script type="module">

alert(this); // undefined

</script>

Есть и несколько других, именно браузерных особенностей скриптов с type="module" по сравнению с обычными скриптами.

Модули всегда выполняются в отложенном (deferred) режиме, точно так же, как скрипты с атрибутом defer. Это верно и для внешних и встроенных скриптов-модулей. Другими словами:

* загрузка внешних модулей, таких как <script type="module" src="...">, не блокирует обработку HTML.
* модули, даже если загрузились быстро, ожидают полной загрузки HTML документа, и только затем выполняются.
* сохраняется относительный порядок скриптов: скрипты, которые идут раньше в документе, выполняются раньше.

Как побочный эффект, модули всегда видят полностью загруженную HTML-страницу, включая элементы под ними. Например:

<script type="module">

alert(typeof button); // object: скрипт может 'видеть' кнопку под ним

</script>

Сравним с обычным скриптом ниже:

<script>

alert(typeof button); // Ошибка: кнопка не определена, скрипт не видит элементы под ним

</script>

<button id="button">Кнопка</button>

Второй скрипт выполнится раньше, чем первый. Поэтому сначала будет выведено undefined, а потом object. Это потому, что модули начинают выполняться после полной загрузки страницы. Обычные скрипты запускаются сразу же, поэтому сообщение из обычного скрипта видно первым.

При использовании модулей стоит иметь в виду, что HTML-страница будет показана браузером до того, как выполнятся модули и JavaScript-приложение будет готово к работе. Некоторые функции могут ещё не работать. Следует разместить «индикатор загрузки» или что-то ещё, чтобы не смутить этим посетителя.

[Атрибут async работает во встроенных скриптах](https://learn.javascript.ru/modules-intro" \l "atribut-async-rabotaet-vo-vstroennyh-skriptah). Для немодульных скриптов атрибут async работает только на внешних скриптах. Скрипты с ним запускаются сразу по готовности, они не ждут другие скрипты или HTML-документ.

Для модулей атрибут async работает на любых скриптах. Например, в скрипте ниже есть async, поэтому он выполнится сразу после загрузки, не ожидая других скриптов. Скрипт выполнит импорт (загрузит ./analytics.js) и сразу запустится, когда будет готов, даже если HTML документ ещё не будет загружен, или если другие скрипты ещё загружаются. Это очень полезно, когда модуль ни с чем не связан, например, для счётчиков, рекламы, обработчиков событий.

<script async type="module">

import {counter} from './analytics.js';

counter.count();

</script>

[**Внешние скрипты**](https://learn.javascript.ru/modules-intro#vneshnie-skripty)

Внешние скрипты с атрибутом type="module" имеют два отличия:

1. Внешние скрипты с одинаковым атрибутом src запускаются только один раз. В примере ниже скрипт my.js загрузится и будет выполнен только один раз:

<script type="module" src="my.js"></script>

<script type="module" src="my.js"></script>

1. Внешний скрипт, который загружается с другого домена, требует указания заголовков [CORS](https://developer.mozilla.org/ru/docs/Web/HTTP/CORS). Другими словами, если модульный скрипт загружается с другого домена, то удалённый сервер должен установить заголовок Access-Control-Allow-Origin означающий, что загрузка скрипта разрешена.

<script type="module" src="http://another-site.com/their.js"></script>

Это обеспечивает лучшую безопасность по умолчанию.

В браузере import должен содержать относительный или абсолютный путь к модулю. Модули без пути называются «голыми» (bare). Они не разрешены в import. Например, этот import неправильный:

import {sayHi} from 'sayHi';

Другие окружения, например, Node.js, допускают использование «голых» модулей, без путей, так как в них есть свои правила, как работать с такими модулями и где их искать. Но браузеры пока не поддерживают «голые» модули.

Старые браузеры не понимают атрибут type="module". Скрипты с неизвестным атрибутом type просто игнорируются. Можно сделать для них «резервный» скрипт при помощи атрибута nomodule:

<script type="module">

alert("Работает в современных браузерах");

</script>

<script nomodule>

alert("Современные браузеры понимают оба атрибута - и type=module, и nomodule, поэтому пропускают этот тег script")

alert("Старые браузеры игнорируют скрипты с неизвестным атрибутом type=module, но выполняют этот.");

</script>

[Инструменты сборки](https://learn.javascript.ru/modules-intro#instrumenty-sborki)

В реальной жизни модули в браузерах редко используются как есть. Обычно, они объединяются вместе, специальными инструментами, например [Webpack](https://webpack.js.org/) и после выкладывается код на рабочий сервер.

Одно из преимуществ использования сборщика – он предоставляет больший контроль над тем, как модули ищутся, позволяет использовать «голые» модули и многое другое «своё», например, CSS/HTML-модули. Сборщик делает следующее:

1. Берёт «основной» модуль, который необходимо поместить в <script type="module"> в HTML.
2. Анализирует зависимости (импорты, импорты импортов и так далее).
3. Собирает один файл со всеми модулями (или несколько файлов, это можно настроить), перезаписывает встроенный import функцией импорта от сборщика, чтобы всё работало. «Специальные» типы модулей, такие как HTML/CSS тоже поддерживаются.
4. В процессе могут происходить и другие трансформации, и оптимизации кода:

* недоступный код удаляется;
* неиспользуемые экспорты удаляются («tree-shaking»);
* специфические операторы для разработки, такие как console и debugger, удаляются;
* современный синтаксис JavaScript также может быть трансформирован в предыдущий стандарт, с похожей функциональностью;
* полученный файл можно минимизировать (удалить пробелы, заменить названия переменных на более короткие и т.д.).

Если используются инструменты сборки, то они объединяют модули вместе в один или несколько файлов, и заменяют import/export на свои вызовы. Поэтому итоговую сборку (в примере ниже – bundle.js) можно подключать и без атрибута type="module", как обычный скрипт:

<script src="bundle.js"></script>

Хотя и «как есть» модули тоже можно использовать, а сборщик настроить позже при необходимости.

1. **Модули: экспорт и импорт**

Директивы экспорт и импорт имеют несколько вариантов вызова.

[**Экспорт до объявления**](https://learn.javascript.ru/import-export#eksport-do-obyavleniya)

Можно пометить любое объявление как экспортируемое, разместив export перед ним, будь то переменная, функция или класс. Например, все следующие экспорты допустимы:

// экспорт массива

export let months = ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

// экспорт константы

export const MODULES\_BECAME\_STANDARD\_YEAR = 2015;

// экспорт класса

export class User {

constructor(name) {

this.name = name;

}

}

Обратите внимание, что export перед классом или функцией не делает их [функциональным выражением](https://learn.javascript.ru/function-expressions-arrows). Это всё также объявление функции, хотя и экспортируемое. Большинство руководств по стилю кода в JavaScript не рекомендуют ставить точку с запятой после объявлений функций или классов. Поэтому в конце export class и export function не нужна точка с запятой:

export function sayHi(user) {

alert(`Hello, ${user}!`);

} // без ; в конце

[**Экспорт отдельно от объявления**](https://learn.javascript.ru/import-export#eksport-otdelno-ot-obyavleniya)

Также можно написать export отдельно. Здесь сначала объявляются функции, а затем экспортируются:

// say.js

function sayHi(user) {

alert(`Hello, ${user}!`);

}

function sayBye(user) {

alert(`Bye, ${user}!`);

}

export {sayHi, sayBye}; // список экспортируемых переменных

Технически также можно расположить export выше функций.

[**Импорт \***](https://learn.javascript.ru/import-export#import)

Обычно список того, что надо импортировать располагается, в фигурных скобках import {...}, например, вот так:

// main.js

import {sayHi, sayBye} from './say.js';

sayHi('John'); // Hello, John!

sayBye('John'); // Bye, John!

Но если импортировать нужно много чего, то можно импортировать всё сразу в виде объекта, используя import \* as <obj>. Например:

// main.js

import \* as say from './say.js';

say.sayHi('John');

say.sayBye('John');

На первый взгляд «импортировать всё» выглядит очень удобно, не надо писать лишнего. Но иногда надо явно перечислять список того, что нужно импортировать , тому есть несколько причин:

1. Современные инструменты сборки ([webpack](http://webpack.github.io/) и другие) собирают модули вместе и оптимизируют их, ускоряя загрузку и удаляя неиспользуемый код.

Предположим, в проект была добавлена сторонняя библиотеку say.js с множеством функций:

// say.js

export function sayHi() { ... }

export function sayBye() { ... }

export function becomeSilent() { ... }

Теперь, если из этой библиотеки в проекте использовать только одну функцию:

// main.js

import {sayHi} from './say.js';

тогда оптимизатор увидит, что другие функции не используются, и удалит остальные из собранного кода, тем самым делая код меньше. Это называется «tree-shaking».

1. Явно перечисляя то, что надо импортировать, можно получить более короткие имена функций: sayHi() вместо say.sayHi().
2. Явное перечисление импортов делает код более понятным, позволяет увидеть, что именно и где используется. Это упрощает поддержку и рефакторинг кода.

**[Импорт «как»](https://learn.javascript.ru/import-export" \l "import-kak)**

Также мжно использовать as, чтобы импортировать под другими именами. Например, для краткости импортируем sayHi в локальную переменную hi, а sayBye импортируем как bye:

// main.js

import {sayHi as hi, sayBye as bye} from './say.js';

hi('John'); // Hello, John!

bye('John'); // Bye, John!

[**Экспортировать «как»**](https://learn.javascript.ru/import-export#eksportirovat-kak)

Аналогичный синтаксис существует и для export. Давайте экспортируем функции, как hi и bye:

// say.js

export {sayHi as hi, sayBye as bye};

Теперь hi и bye – официальные имена для внешнего кода, их нужно использовать при импорте:

// main.js

import \* as say from './say.js';

say.hi('John'); // Hello, John!

say.bye('John'); // Bye, John!

[**Экспорт по умолчанию**](https://learn.javascript.ru/import-export#eksport-po-umolchaniyu)

На практике модули встречаются в основном одного из двух типов:

1. Модуль, содержащий библиотеку или набор функций, как say.js выше.
2. Модуль, который объявляет что-то одно, например, модуль user.js экспортирует только class User.

По большей части, удобнее второй подход, когда каждая «вещь» находится в своём собственном модуле. Естественно, требуется много файлов, если для всего делать отдельный модуль, но это не проблема. Так удобнее: навигация по проекту становится проще, особенно, если у файлов хорошие имена, и они структурированы по папкам.

Модули предоставляют специальный синтаксис export default («эспорт по умолчанию») для второго подхода. Указываем export default перед тем, что нужно экспортировать:

// user.js

export default class User {

constructor(name) {

this.name = name;

}

}

В файле может быть не более одного export default, и потом импорт без фигурных скобок:

// main.js

import User from './user.js'; // не {User}, просто User

new User('John');

Импорты без фигурных скобок выглядят красивее. Обычная ошибка начинающих: забывать про фигурные скобки. Запомним: фигурные скобки необходимы в случае именованных экспортов, для export default они не нужны.

| **Именованный экспорт** | **Экспорт по умолчанию** |
| --- | --- |
| export class User {...} | export default class User {...} |
| import {User} from ... | import User from ... |

Технически в одном модуле может быть, как экспорт по умолчанию, так и именованные экспорты, но на практике обычно их не смешивают. То есть, в модуле находятся либо именованные экспорты, либо один экспорт по умолчанию.

Так как в файле может быть максимум один export default, то экспортируемая сущность не обязана иметь имя. Например, всё это – полностью корректные экспорты по умолчанию:

export default class { // у класса нет имени

constructor() { ... }

}

export default function(user) { // у функции нет имени

alert(`Hello, ${user}!`);

}

// экспортируем значение, не создавая переменную

export default ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

Это нормально, потому что может быть только один export default в файле, так что import без фигурных скобок всегда знает, что импортировать. Без default такой экспорт выдал бы ошибку, так как необходимо имя класса, если это не экспорт по умолчанию:

export class { // Ошибка

constructor() {}

}

[**Имя «default»**](https://learn.javascript.ru/import-export#imya-default)

В некоторых ситуациях для обозначения экспорта по умолчанию в качестве имени используется default. Например, чтобы экспортировать функцию отдельно от её объявления:

function sayHi(user) {

alert(`Hello, ${user}!`);

}

export {sayHi as default};

Представим следующую ситуацию: модуль user.js экспортирует одну сущность «по умолчанию» и несколько именованных:

// user.js

export default class User {

constructor(name) {

this.name = name;

}

}

export function sayHi(user) {

alert(`Hello, ${user}!`);

}

Вот как импортировать экспорт по умолчанию вместе с именованным экспортом:

// main.js

import {default as User, sayHi} from './user.js';

new User('John');

Если импортируется всё как объект import \*, тогда его свойство default – как раз и будет экспортом по умолчанию:

// main.js

import \* as user from './user.js';

let User = user.default; // экспорт по умолчанию

new User('John');

[Однако, у экспорта по умолчанию](https://learn.javascript.ru/import-export#dovod-protiv-eksportov-po-umolchaniyu) есть недостатки. Именованные экспорты «включают в себя» своё имя. Эта информация является частью модуля, она сообщает, что именно экспортируется. Именованные экспорты вынуждают использовать правильное имя при импорте:

// import {MyUser} не сработает, должно быть именно имя {User}

import {User} from './user.js';

В то время как для экспорта по умолчанию можно выбрать любое имя при импорте:

import User from './user.js'; // сработает

import MyUser from './user.js'; // сработает

Так что члены команды разработчиков могут использовать разные имена для импорта одной и той же вещи, и это не очень хорошо. Обычно, чтобы избежать этого и соблюсти единообразие кода, есть правило: имена импортируемых переменных должны соответствовать именам файлов. Вот так:

import User from './user.js';

import LoginForm from './loginForm.js';

import func from '/path/to/func.js';

Тем не менее, в некоторых командах это считают серьёзным доводом против экспортов по умолчанию и предпочитают использовать именованные экспорты везде. Даже если экспортируется только одна вещь, она всё равно экспортируется с именем, без использования default. Это также немного упрощает реэкспорт.

[**Реэкспорт**](https://learn.javascript.ru/import-export#reeksport)

Синтаксис «реэкспорта» export ... from ... позволяет импортировать что-то и тут же экспортировать, даже под другим именем:

export {sayHi} from './say.js'; // реэкспортировать sayHi

export {default as User} from './user.js'; // реэкспортировать default

Рассмотрим практический пример использования. Представим, что разрабатывается «пакет»: папка со множеством модулей, из которой часть функционала экспортируется наружу (инструменты вроде NPM позволяют публиковать и распространять такие пакеты), а многие модули – просто вспомогательные, для внутреннего использования в других модулях пакета. Структура файлов может быть такой:

auth/

index.js

user.js

helpers.js

tests/

login.js

providers/

github.js

facebook.js

...

Надо сделать функционал пакета доступным через единую точку входа: «главный файл» auth/index.js. Чтобы можно было использовать его следующим образом:

import {login, logout} from 'auth/index.js'

Идея в том, что внешние разработчики, которые будут использовать пакет, не должны разбираться с его внутренней структурой. Всё, что нужно, надо экспортировать в auth/index.js, а остальное скрыть от разработчиков. Так как нужный функционал может быть разбросан по модулям пакета, то надо импортировать их в auth/index.js и тут же экспортировать наружу.

// auth/index.js

// импортировать login/logout и тут же экспортировать

import {login, logout} from './helpers.js';

export {login, logout};

// импортировать экспорт по умолчанию как User и тут же экспортировать

import User from './user.js';

export {User};

Теперь пользователи пакета могут писать import {login} from "auth/index.js". Запись export ... from ...– это просто более короткий вариант такого импорта-экспорта:

// auth/index.js

// импортировать login/logout и тут же экспортировать

export {login, logout} from './helpers.js';

// импортировать экспорт по умолчанию как User и тут же экспортировать

export {default as User} from './user.js';

[**Реэкспорт экспорта по умолчанию**](https://learn.javascript.ru/import-export#reeksport-eksporta-po-umolchaniyu)

При реэкспорте экспорт по умолчанию нужно обрабатывать особым образом. Например, есть user.js, из которого надо реэкспортировать класс User:

// user.js

export default class User {

// ...

}

1. export User from './user.js' не будет работать, возникнет синтаксическая ошибка.

Чтобы реэкспортировать экспорт по умолчанию, надо написать export {default as User}, как в примере выше.

1. export \* from './user.js' реэкспортирует только именованные экспорты, исключая экспорт по умолчанию.

Если надо реэкспортировать и именованные экспорты, и экспорт по умолчанию, то понадобятся две инструкции:

export \* from './user.js'; // для реэкспорта именованных экспортов

export {default} from './user.js'; // для реэкспорта по умолчанию

Такое особое поведение реэкспорта с экспортом по умолчанию – одна из причин того, почему их неудобно использовать.

1. **Модули: динамические импорты**

Инструкции экспорта и импорта, которые рассматривались в предыдущем вопросе, называются «статическими». Синтаксис у них весьма простой и строгий.

Во-первых, нельзя динамически задавать никакие из параметров import. Путь к модулю должен быть строковым примитивом и не может быть вызовом функции. Вот так работать не будет:

import ... from getModuleName(); // Ошибка, должна быть строка

Во-вторых, нельзя делать импорт в зависимости от условий или в процессе выполнения.

if(...) {

import ...; // Ошибка, запрещено

}

{

import ...; // Ошибка, нельзя ставить импорт в блок

}

Всё это следствие того, что цель директив import/export – задать костяк структуры кода. Благодаря им она может быть проанализирована, модули могут быть собраны в один файл специальными инструментами, а неиспользуемые экспорты удалены. Это возможно только благодаря тому, что всё статично.

[**Выражение import()**](https://learn.javascript.ru/modules-dynamic-imports#vyrazhenie-import)

Выражение import(module) загружает модуль и возвращает промис, результатом которого становится объект модуля, содержащий все его экспорты. Использовать его можно динамически в любом месте кода, например, так:

let modulePath = prompt("Какой модуль загружать?");

import(modulePath)

.then(obj => <объект модуля>)

.catch(err => <ошибка загрузки, например если нет такого модуля>)

Или если внутри асинхронной функции, то можно let module = await import(modulePath). Например, если у нас есть такой модуль say.js:

// say.js

export function hi() {

alert(`Привет`);

}

export function bye() {

alert(`Пока`);

}

То динамический импорт может выглядеть так:

let {hi, bye} = await import('./say.js');

hi();

bye();

А если в say.js указан экспорт по умолчанию:

// say.js

export default function() {

alert("Module loaded (export default)!");

}

То для доступа к нему следует взять свойство default объекта модуля:

let obj = await import('./say.js');

let say = obj.default;

// или, одной строкой: let {default: say} = await import('./say.js');

say();

Вот полный пример:



// say.js

export function hi() {

alert(`Привет`);

}

export function bye() {

alert(`Пока`);

}

export default function() {

alert("Модуль загружен (экспорт по умолчанию)!");

}

// index.html

<!doctype html>

<script>

async function load() {

let say = await import('./say.js');

say.hi(); // Привет!

say.bye(); // Пока!

say.default(); // Модуль загружен (экспорт по умолчанию)!

}

</script>

<button onclick="load()">Нажми меня</button>

Динамический импорт работает в обычных скриптах, он не требует указания script type="module".

Хотя import() и выглядит похоже на вызов функции, на самом деле это специальный синтаксис, так же, как, например, super().

Так что нельзя копировать import в другую переменную или вызвать при помощи .call/apply. Это не функция.