# Типы данных

В JavaScript есть 8 основных типов данных.

* Семь из них называют «**примитивными**» типами данных:
  + number для любых чисел: целочисленных или чисел с плавающей точкой; целочисленные значения ограничены диапазоном ±(253-1).
  + bigint для целых чисел произвольной длины.
  + string для строк. Строка может содержать ноль или больше символов, нет отдельного символьного типа.
  + boolean для true/false.
  + null для неизвестных значений – отдельный тип, имеющий одно значение null. (не имеют объектов-обёрток, прототипов)
  + undefined для неприсвоенных значений – отдельный тип, имеющий одно значение undefined. (не имеют объектов-обёрток, прототипов)
  + symbol для уникальных идентификаторов.
* И один не является «примитивным» и стоит особняком:
  + object для более сложных структур данных.

Оператор typeof позволяет нам увидеть, какой тип данных сохранён в переменной.

* Имеет две формы: typeof x или typeof(x).
* Возвращает строку с именем типа. Например, "string".
* Для null возвращается "object" – это ошибка в языке, на самом деле это не объект.

# Объекты

**Создание**

let user = { name: "John" };

let admin = { name: "Admin" };

**Доступ к свойству**

user.name или user[“name”]

**Добавление функции в объект**

user.f = sayHi;

admin.f = sayHi;

**Вызов функции**

user.f(); // John (this == user)

admin.f(); // Admin (this == admin)

admin['f'](); // Admin (нет разницы между использованием точки или квадратных скобок для доступа к объекту)

В JavaScript this является «свободным», его значение вычисляется в момент вызова метода и не зависит от того, где этот метод был объявлен, а скорее от того, какой объект вызывает метод (какой объект стоит «перед точкой»).

# Функции

|  |  |
| --- | --- |
| *Function Declaration* (Объявление Функции) | *Function Expression* (Функциональное Выражение). |
| function sayHi() {  alert( "Привет" );  } | let sayHi = function() {  alert( "Привет" );  }; |
| Function Declaration может быть вызвана раньше, чем она объявлена.  Function Declaration находится в блоке {...}, функция доступна везде внутри блока. Но не снаружи него. Функции, объявленные таким образом, можно вызывать до их объявления. | Function Expression создаётся, когда выполнение доходит до него, и затем уже может использоваться. |

Функция – это значение, **представляющее «действие».** Они могут быть присвоены, скопированы или объявлены в любом месте кода.

|  |
| --- |
| function sayHi() { //создаёт функцию и помещает её в переменную с именем sayHi  alert( "Привет" );  }  alert( sayHi ); // выведет код функции, но не вызовет ее  let func = sayHi; // копируем  func(); // Привет // вызываем копию (работает)!  sayHi(); // Привет // эта тоже все ещё работает (почему бы и нет) |

## [Функции-«колбэки»](https://learn.javascript.ru/function-expressions" \l "funktsii-kolbeki)

|  |
| --- |
| function ask(question, yes, no) {  if (confirm(question)) yes()  else no();  }  function showOk() {  alert( "Вы согласны." );  }  function showCancel() {  alert( "Вы отменили выполнение." );  }  // использование: функции showOk, showCancel передаются в качестве аргументов ask  ask("Вы согласны?", showOk, showCancel); |

Аргументы showOk и showCancel функции ask называются функциями-колбэками или просто колбэками.

Можно переписать так:

|  |
| --- |
| function ask(question, yes, no) {  if (confirm(question)) yes()  else no();  }  ask(  "Вы согласны?",  function() { alert("Вы согласились."); },  function() { alert("Вы отменили выполнение."); }  ); |

Здесь функции объявляются прямо внутри вызова ask(...). У них нет имён, поэтому они называются *анонимными*. Такие функции недоступны снаружи ask (потому что они не присвоены переменным).

## Стрелочные функции

|  |  |
| --- | --- |
| let func = function(arg1, arg2, ...argN) {  return expression;  }; | let func = (arg1, arg2, ...argN) => expression; |
|  | let sum = (a, b) => { // фигурная скобка, открывающая тело многострочной функции  let result = a + b;  return result; // если мы используем фигурные скобки, то нам нужно явно указать "return"  };  alert( sum(1, 2) ); // 3 |

## Прозрачное кеширование. Декоратор

Функция slow(x), выполняющая ресурсоёмкие вычисления, но возвращающая стабильные результаты. Другими словами, для одного и того же x она всегда возвращает один и тот же результат.

*Декоратор* - специальная функция, которая принимает другую функцию и изменяет её поведение.

|  |
| --- |
| function slow(x) {  // здесь могут быть ресурсоёмкие вычисления  alert(`Called with ${x}`);  return x;  }  function cachingDecorator(func) {  let cache = new Map();  return function(x) {  if (cache.has(x)) { // если кеш содержит такой x,  return cache.get(x); // читаем из него результат  }  let result = func(x); // иначе, вызываем функцию  cache.set(x, result); // и кешируем (запоминаем) результат  return result;  };  }  slow = cachingDecorator(slow);  alert( slow(1) ); // slow(1) кешируем  alert( "Again: " + slow(1) ); // возвращаем из кеша  alert( slow(2) ); // slow(2) кешируем  alert( "Again: " + slow(2) ); // возвращаем из кеша |

## [Применение «func.call» для передачи контекста.](https://learn.javascript.ru/call-apply-decorators" \l "primenenie-func-call-dlya-peredachi-konteksta)

Упомянутый выше кеширующий декоратор не подходит для работы с методами объектов.

|  |
| --- |
| function sayHi() {  alert(this.name);  }  let user = { name: "John" };  let admin = { name: "Admin" };  // используем 'call' для передачи различных объектов в качестве 'this'  sayHi.call( user ); // John  sayHi.call( admin ); // Admin |

|  |
| --- |
| let worker = {  someMethod() {  return 1;  },  slow(x) {  alert("Called with " + x);  return x \* this.someMethod(); // (\*)  }  };  function cachingDecorator(func) {  let cache = new Map();  return function(x) {  if (cache.has(x)) {  return cache.get(x);  }  let result = **func.call**(this, x); // теперь 'this' передаётся правильно  cache.set(x, result);  return result;  };  }  worker.slow = cachingDecorator(worker.slow); // теперь сделаем её кеширующей  alert( worker.slow(2) ); // работает  alert( worker.slow(2) ); // работает, не вызывая первоначальную функцию (кешируется) |

C несколькими аргументами функции

|  |  |  |
| --- | --- | --- |
| let worker = {  slow(min, max) {  alert(`Called with ${min},${max}`);  return min + max;  }  };  function cachingDecorator(func, hash) {  let cache = new Map();  return function() {  let key = hash(arguments); // (\*)  if (cache.has(key)) {  return cache.get(key);  } | | |
| let result = func.call(this, ...arguments); // (\*\*) | | let result = func.apply(context, args); |
| cache.set(key, result);  return result;  };  } | | |
| function hash(args) {  return args[0] + ',' + args[1];  } | function hash() {  alert( [].join.call(arguments) );  }  //заимствование метода | |
| worker.slow = cachingDecorator(worker.slow, hash);  alert( worker.slow(3, 5) ); // работает  alert( "Again " + worker.slow(3, 5) ); // аналогично (из кеша) | | |

## [Переходим к нескольким аргументам с «func.apply»](https://learn.javascript.ru/call-apply-decorators" \l "perehodim-k-neskolkim-argumentam-s-func-apply)

Единственная разница в синтаксисе между call и apply состоит в том, что call ожидает список аргументов, в то время как apply принимает псевдомассив.

Эти два вызова почти эквивалентны:

func.call(context, ...args); // передаёт массив как список с оператором расширения

func.apply(context, args); // тот же эффект

Есть только одна небольшая разница:

* Оператор расширения ... позволяет передавать *перебираемый* объект args в виде списка в call.
* А apply принимает только *псевдомассив* args.

|  |
| --- |
| var obj = { num: 2 };  function add(a, b){ return this.num + a + b; }  console.log(add.apply(obj, [3, 5])); |

## [Метод «call» для передачи контекста.](https://learn.javascript.ru/call-apply-decorators" \l "primenenie-func-call-dlya-peredachi-konteksta)

[Применение методов Bind(), Call(), and Apply() в JavaScript (nuancesprog.ru)](https://nuancesprog.ru/p/13559/?ysclid=lx1os8l1oz275559155)

Метод call() вызывает функцию с заданным контекстом. Иначе говоря, вы привязываете функцию к объекту, как если бы она ему принадлежала.

|  |
| --- |
| var obj = { num: 2 };  function add(a){ return this.num + a; }  add.call(obj, 3); |
| var obj = { num: 2 };  function add(a, b){ return this.num + a + b; }  console.log(add.call(obj, 3, 5)); |

* Теперь функция add() получает свой **this**из obj, к которому она привязана.
* При вызове функции add() this.num ссылается на num объекта obj.
* В результате вызов возвращает 5, поскольку 2 + 3 = 5.

# Модули

ссылка: [Модули, введение (javascript.ru)](https://learn.javascript.ru/modules-intro)

Модуль – это просто файл. Один скрипт – это один модуль.

Модули не работают локально. Только через HTTP(s).

export отмечает переменные и функции, которые должны быть доступны вне текущего модуля.

import позволяет импортировать функциональность из других модулей.

|  |  |
| --- | --- |
| // 📁 sayHi.js | // 📁 main.js |
| export function sayHi(user) {  alert(`Hello, ${user}!`);  } | import {sayHi} from './sayHi.js';  alert(sayHi); // function...  sayHi('John'); // Hello, John! |

При использовании непосредственно в html необходимо казать браузеру, что скрипт является модулем, при помощи атрибута <script type="module">.

|  |
| --- |
| <!doctype html>  <script type="module">  import {sayHi} from './say.js';  document.body.innerHTML = sayHi('John');  </script> |

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

|  |  |
| --- | --- |
| **hello.js** | **user.js** |
| let user = "John"; | alert(user); // в этом модуле нет такой переменной (каждый модуль имеет независимые переменные) |
| **index.html** |  |
| <!doctype html>  <script type="module" src="user.js"></script>  <script type="module" src="hello.js"></script> |  |

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 нам нужно сделать глобальную переменную уровня всей страницы, можно явно присвоить её объекту window, тогда получить значение переменной можно обратившись к window.user.

[**Код в модуле выполняется только один раз при импорте**](https://learn.javascript.ru/modules-intro#kod-v-module-vypolnyaetsya-tolko-odin-raz-pri-importe)**:** если один и тот же модуль используется в нескольких местах, то его код выполнится только один раз.

Объект import.meta содержит информацию о текущем модуле:

|  |
| --- |
| <script type="module">  alert(import.meta.url); // ссылка на html страницу для встроенного скрипта  </script> |

[В модуле «this» не определён](https://learn.javascript.ru/modules-intro" \l "v-module-this-ne-opredelyon)

|  |
| --- |
| <script>  alert(this); // window  </script>  <script type="module">  alert(this); // undefined  </script> |

[Особенности в браузерах](https://learn.javascript.ru/modules-intro" \l "osobennosti-v-brauzerah)

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

Модули всегда выполняются в отложенном (deferred) режиме:

* 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.

|  |
| --- |
| <script type="module">  alert(typeof button); // object: скрипт может 'видеть' кнопку под ним  // так как модули являются отложенными, то скрипт начнёт выполнятся только после полной загрузки страницы  </script>  Сравните с обычным скриптом ниже:  <script>  alert(typeof button); // Ошибка: кнопка не определена, скрипт не видит элементы под ним  // обычные скрипты запускаются сразу, не дожидаясь полной загрузки страницы  </script>  <button id="button">Кнопка</button> |

Внимание: второй скрипт выполнится раньше, чем первый! Поэтому мы увидим сначала undefined, а потом object Это потому, что модули начинают выполняться после полной загрузки страницы. Обычные скрипты запускаются сразу же, поэтому сообщение из обычного скрипта мы видим первым.

[Атрибут async работает во встроенных скриптах](https://learn.javascript.ru/modules-intro" \l "atribut-async-rabotaet-vo-vstroennyh-skriptah)

Для не-модульных 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.

|  |
| --- |
| <!-- загружаются зависимости (analytics.js) и скрипт запускается -->  <!-- модуль не ожидает загрузки документа или других тэгов <script> -->  <script async type="module">  import {counter} from './analytics.js';  counter.count();  </script> |

Это очень полезно, когда модуль ни с чем не связан, например для счётчиков, рекламы, обработчиков событий.

[Не допускаются «голые» модули](https://learn.javascript.ru/modules-intro" \l "ne-dopuskayutsya-golye-moduli)

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

|  |
| --- |
| import {sayHi} from 'sayHi'; // Ошибка, "голый" модуль  // путь должен быть, например './sayHi.js' или абсолютный |

Совместимость, «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> |

# Экспорт и импорт

## Экспорт

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

Мы можем пометить любое объявление как экспортируемое, разместив export перед ним, будь то переменная, функция или класс.

|  |
| --- |
| // экспорт массива  export let months = ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']; |

**Не ставится точка с запятой после экспорта класса/функции**

|  |
| --- |
| export function sayHi(user) {  alert(`Hello, ${user}!`);  } // без ; в конце |

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

|  |
| --- |
| // 📁 say.js  function sayHi(user) {  alert(`Hello, ${user}!`);  }  function sayBye(user) {  alert(`Bye, ${user}!`);  }  export {sayHi, sayBye}; // список экспортируемых переменных |

Самое важное

Перед объявлением класса/функции/…:

export [default] class/function/variable ...

Отдельный экспорт:

export {x [as y], ...}.

Реэкспорт:

export {x [as y], ...} from "module"

export \* from "module" (не реэкспортирует export default).

export {default [as y]} from "module" (реэкспортирует только export default).

## Импорт

Лучше использовать явное перечисление импортов:

* вставляет только нужную функцию;
* получаем более короткие имена функций.

Располагаем список того, что хотим импортировать, в фигурных скобках import {...}, например вот так:

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

Если импортировать нужно много чего, мы можем импортировать всё сразу в виде объекта, используя import \* as <obj>

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

[Импорт «как»](https://learn.javascript.ru/import-export" \l "import-kak). Мы также можем использовать as, чтобы импортировать под другими именами.

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

[Экспортировать «как»](https://learn.javascript.ru/import-export" \l "eksportirovat-kak). Теперь hi и bye – официальные имена для внешнего кода, их нужно использовать при импорте:

|  |
| --- |
| export {sayHi as hi, sayBye as bye}; |
| В другом файле  import \* as say from './say.js';  say.hi('John'); // Hello, John!  say.bye('John'); // Bye, John! |

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.

Модули предоставляют специальный синтаксис export default («экспорт по умолчанию») для второго подхода. В файле может быть не более одного export default. И потом импортируем без фигурных скобок.

|  |  |
| --- | --- |
| // 📁 user.js  export default class User { // просто добавьте "default"  constructor(name) {  this.name = name;  }  } | // 📁 main.js  import User from './user.js'; // не {User}, просто User  new User('John'); |

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

Так как в файле может быть максимум один export default, то экспортируемая сущность не обязана иметь имя.

|  |
| --- |
| // экспортируем значение, не создавая переменную  export default ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']; |

[Имя «default»](https://learn.javascript.ru/import-export" \l "imya-default)

В некоторых ситуациях для обозначения экспорта по умолчанию в качестве имени используется default.

|  |
| --- |
| function sayHi(user) {  alert(`Hello, ${user}!`);  }  // то же самое, как если бы мы добавили "export default" перед функцией  export {sayHi as default}; |

 Одна сущность «по умолчанию» и несколько именованных (редкий, но возможный случай):

|  |  |
| --- | --- |
| // 📁 user.js  export default class User {  constructor(name) {  this.name = name;  }  }  export function sayHi(user) {  alert(`Hello, ${user}!`);  } | // 📁 main.js  import {default as User, sayHi} from './user.js';  new User('John'); |

Импортируем 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'); |

Довод против экспортов по умолчанию:

* именованные экспорты вынуждают нас использовать правильное имя при импорте:
* для экспорта по умолчанию мы выбираем любое имя при импорте, поэтому желательно, чтобы имена импортируемых переменных соответствовали именам файлов.

Самое важное

Импорт:

Именованные экспорты из модуля:

import {x [as y], ...} from "module"

Импорт по умолчанию:

import x from "module"

import {default as x} from "module"

Всё сразу:

import \* as obj from "module"

Только подключить модуль (его код запустится), но не присваивать его переменной:

import "module"

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

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

|  |
| --- |
| export {sayHi} from './say.js'; // реэкспортировать sayHi  export {default as User} from './user.js'; // реэкспортировать default |

Это нужно для того, чтобы в последствии можно было все импортировать из одного файла.

Так как нужная функциональность может быть разбросана по модулям нашего пакета, мы можем 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 нашего 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'; |

Самое важное

Перед объявлением класса/функции/…: export [default] class/function/variable ...

Отдельный экспорт: export {x [as y], ...}.

Реэкспорт:

export {x [as y], ...} from "module"

export \* from "module" (не реэкспортирует export default).

export {default [as y]} from "module" (реэкспортирует только export default).

# Выбор и изменение элементов страницы

let button = document.querySelector(".button");

button.textContent = “Какой-то текст”

let advice = document.querySelector('.advice');

advice.style.fontSize = '33px';

# Реакция на действия пользователя

let button = document.querySelector('.button');

button.addEventListener('click', function () {

alert(“hello”);

});

Массив

let user = ['Мария', 21, 'Железногорск'];

Элементом массива может быть и объект.

Объект

Объекты содержат свойства. Свойство объекта состоит из ключа, например name, и значения по этому ключу — 'Мария'.

let user = {

name: 'Мария',

dotaLevel: 21,

dogName: 'Железногорск'

};

# CSS

/\* Селектор можно перевести как «после зажатия кнопки» \*/

button:active::after {

animation: change-bg 10s step-end infinite;

}

# Проверка существует ли свойство (опциональная цепочка?.)

Опциональная цепочка ?. останавливает выполнение и возвращает undefined, если значение перед ?. равно undefined или null.

user?.var – существует ли переменная

user.function?.() – существует ли функция

user?.[key] – существует ли свойство объекта

# Регулярные выражения

«Длинный» синтаксис: regexp = new RegExp("шаблон", "флаги");

Короткий синтаксис, использующий слеши "/":

regexp = /шаблон/; // без флагов

regexp = /шаблон/gmi; // с флагами gmi (будут описаны далее)

Основная разница между этими двумя способами создания заключается в том, что слеши /.../ не допускают никаких вставок переменных (наподобие возможных в строках через ${...}). Они полностью статичны.

Слеши используются, когда мы на момент написания кода точно знаем, каким будет регулярное выражение – и это большинство ситуаций. А new RegExp – когда мы хотим создать регулярное выражение «на лету» из динамически сгенерированной строки, например:

let tag = prompt("Какой тег вы хотите найти?", "h2");

let regexp = new RegExp(`<${tag}>`);

## Флаги:

i - поиск не зависит от регистра

g - поиск ищет все совпадения, без него – только первое

m – многострочный режим, влияет на поведение якорей

s - режим «dotall», при котором точка. может соответствовать символу перевода строки \n

## Функции

Метод **str.match(regexp)** для строки str возвращает совпадения с регулярным выражением regexp.

Метод **str.replace(regexp, replacement)** заменяет совпадения с regexp в строке str на replacement (все, если есть флаг g, иначе только первое).

Метод **regexp.test(str)** проверяет, есть ли хоть одно совпадение, если да, то возвращает true, иначе false.

## Символьные классы

Символьный класс – это специальное обозначение, которое соответствует любому символу из определённого набора.

\d – любая цифра от 0 до 9

\s - пробельные символы: включает в себя символ пробела, табуляции \t, перевода строки \n и некоторые другие редкие пробельные символы, обозначаемые как \v, \f и \r

\w - символ «слова», а точнее – буква латинского алфавита или цифра или подчёркивание \_. Нелатинские буквы не являются частью класса \w, то есть буква русского алфавита не подходит.

Точка . – это специальный символьный класс, который соответствует «любому символу, кроме новой строки».

## [Обратные символьные классы](https://learn.javascript.ru/regexp-character-classes" \l "obratnye-simvolnye-klassy)

Для каждого символьного класса существует «обратный класс», обозначаемый той же буквой, но в верхнем регистре. «Обратный» означает, что он соответствует всем другим символам.

«Обратный» означает, что он соответствует всем другим символам, например:

\D - не цифра: любой символ, кроме \d, например буква.

\S - не пробел: любой символ, кроме \s, например буква.

\W - любой символ, кроме \w, то есть не буквы из латиницы, не знак подчёркивания и не цифра. В частности, русские буквы принадлежат этому классу.

## Якоря

Каретка ^ означает совпадение с началом текста, а доллар $ – с концом.

let str1 = "Mary had a little lamb";

alert( /^Mary/.test(str1) ); // true

## Граница слова \b

Есть три вида позиций, которые являются границами слова:

* начало текста, если его первый символ \w.
* позиция внутри текста, если слева находится \w, а справа – не \w, или наоборот.
* конец текста, если его последний символ \w.

## Экранирование символов

Для поиска специальных символов [ ] \ ^ $ . | ? \* + ( ), нам нужно добавить перед ними \ («экранировать их»).

При создании через new RegExpr удвоить обратную косую черту, например \\.

## Наборы

Для примера, [eao] означает любой из 3-х символов: 'a', 'e' или 'o'.

## [Диапазоны](https://learn.javascript.ru/regexp-character-sets-and-ranges" \l "diapazony)

К примеру, [a-z] соответствует символу в диапазоне от a до z, или [0-5] – цифра от 0 до 5.

в [0-9A-F] сразу два диапазона: ищется символ, который либо цифра от 0 до 9, либо буква от A до F.

## [Исключающие диапазоны](https://learn.javascript.ru/regexp-character-sets-and-ranges#isklyuchayuschie-diapazony)

[^aeyo] – любой символ, за исключением 'a', 'e', 'y' или 'o'.

В квадратных скобках большинство специальных символов можно использовать без экранирования:

Символы . + ( ) не нужно экранировать никогда.

Тире - не надо экранировать в начале или в конце (где оно не задаёт диапазон).

Символ каретки ^ нужно экранировать только в начале (где он означает исключение).

Закрывающую квадратную скобку ], если нужен именно такой символ, экранировать нужно.

Другими словами, разрешены без экранирования все специальные символы, кроме случаев, когда они означают что-то особое в наборах.

Точка . внутри квадратных скобок – просто точка. Шаблон [.,] будет искать один из символов: точку или запятую.

## Квантификаторы

**Количество {n}**

Шаблон \d{5} обозначает ровно 5 цифр, он эквивалентен \d\d\d\d\d.

Мы можем добавить \b, чтобы исключить числа длиннее: \b\d{5}\b.

**Диапазон: {3,5}, от 3 до 5**

Для того, чтобы найти числа от 3 до 5 цифр, мы можем указать границы в фигурных скобках: \d{3,5}

Шаблон \d{3,} найдёт последовательность чисел длиной 3 и более цифр

**[Короткие обозначения](https://learn.javascript.ru/regexp-quantifiers" \l "korotkie-oboznacheniya)**

+ - означает «один или более». То же самое, что и {1,}.

? - означает «ноль или один». То же самое, что и {0,1}.

\* - означает «ноль или более». То же самое, что и {0,}

### Жадные

Движок регулярного выражения пытается получить максимальное количество символов, соответствующих .+, а затем сокращает это количество символ за символом, если остаток шаблона не совпадает.

### Ленивые

# Прототипное наследование

let animal = {

eats: true

};

let rabbit = {

jumps: true,

\_\_proto\_\_: animal

};

// Object.keys возвращает только собственные ключи

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

// for..in проходит и по своим, и по унаследованным ключам

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

Посмотреть в консоли цепочку наследования

console.dir([1,2,3])