**Регулярные выражения. История**

Вы уже знаете, что у input есть атрибут pattern, в который записывают регулярное выражение. Но об этом не расскажешь в одном абзаце. Так что мы написали целую тему.

История регулярных выражений началась с дружбы американского логика Уолтера Питтса и физиолога Уоррена МакКаллока.

Вместе эти два человека создали первую механистическую теорию сознания, первые математические модели нейрона, разработали компьютерную логику и стали основоположниками теории искусственного интеллекта. Но обо всём по-порядку.

В 1936 году Алан Тьюринг опубликовал статью, в которой доказал, что машина способна произвести математическое вычисление. МакКаллок прочитал эту работу и понял, что наш мозг работает подобно машине Тьюринга: использует логику нейросетей для произведения вычислений. Он полагал, что нейроны связаны друг с другом по законам формальной логики, и эти связи выстраивают сложнейшие мыслительные цепочки.

МакКаллок поделился своими наблюдениями с Питтсом, и результатом этого творческого союза стал нейрон МакКаллока-Питтса.

Нейроны объединяются в сети, которые называются нейронными сетями МакКаллока-Питтса. Вся сеть функционирует как конечный автомат: число возможных внутренних состояний конечно.

Американский математик Стивен Клини взял за основу открытия МакКаллока и Питтса и представил миру работу «Представление событий в нервных сетях и конечных автоматах». В ней он предложил способ описания событий, которые происходят внутри нейронной сети. Мы описываем происходящие в сетях тремя операторами: \*, | и оператором конкатенации. Ими можно описать события и в сети МакКаллока-Питтса. Такие события Клини назвал регулярными (в том смысле, что других событий не может быть).

Всё это имело строго теоретический характер. Но затем изобрели транзистор, и теория начала века стала применятся в физических устройствах — первых компьютерах. Примерно через 15 лет после публикации статьи Клини, в 1960 году, американский программист Кен Томпсон обнаружил, что регулярными выражениями удобно создавать шаблоны для поиска строк в длинных текстах. Регулярное выражение преобразуется в конечный автомат и производит поиск строк, соответствующих шаблону. Томпсон включил поиск по регулярным выражениям в свой текстовый редактор ed, и с тех пор этот способ стал стандартом.

Люди разных профессий, вооружившись теоретической математикой и используя труды своих современников, сформулировали теорию о том, как мы мыслим и работаем со своей памятью, а программисты вдохновились этими идеями и создали удобный инструмент.

Мы погрузимся в практическую часть — Regular Expression (англ. «регулярные выражения»), RegExp, RegEx или просто «Регулярку».

# Регулярные выражения в JS

Регулярные выражения — отдельный язык программирования. Его создали для поиска и замены части текста. Например, если в тексте допущена ошибка в одном повторяющемся слове, можно найти и исправить все такие ошибки за раз.

Работать с регулярными выражениями можно на разных языках: Java, C++, PHP, Python и, конечно, JavaScript.

## Как работают регулярные выражения

Первым делом нужно создать шаблон регулярного выражения — алгоритм, по которому нужно искать строку в тексте. Например: нужно искать кириллические заглавные буквы «С».

Дальше шаблон передают специальным методам, которые ищут, заменяют и удаляют нужные символы. Таким образом, шаблон определяет, что и как искать, а метод — что с этим потом делать.

## Как создать регулярное выражение

Чтобы создать регулярное выражение, нужно заключить шаблон в слеши: /.../. Символы зарезервированы для этого в JS. Разработчики называют /.../ «литералами регулярного выражения»:

Скопировать кодJAVASCRIPT

const regex = /регулярно выражаюсь/;

console.log(regex); *// /регулярно выражаюсь/*

## Как использовать регулярное выражение

Простой пример — найдём в строке Иннокентия:

Скопировать кодJAVASCRIPT

const userList = 'Маша, Петя, Катя, Лёша, Лера, Иннокентий, Влад';

const innokentiy = /Иннокентий/g;

userList.match(innokentiy); *// [ "Иннокентий" ]*

Переменная innokentiy содержит шаблон регулярного выражения. По нему можно найти, удалить или заменить слово «Иннокентий» в любом тексте. На символ g пока не обращайте внимания: скоро разберём, что это.

Что именно нужно сделать с символом, определяет метод. Здесь мы вызвали match: он найдёт Иннокентия в тексте и завершит работу.

Конечно, регулярные выражения позволяют писать куда более сложные алгоритмы поиска. Как это делается, расскажем дальше в теме.

## Применим на практике

Предположим, вы написали статью со сравнением моделей телефонов. В конце попросили читателей оставить комментарий, у кого какой смартфон. Каждый пишет отзыв, как хочет:

Скопировать кодJAVASCRIPT

const comments = [

'Я пользуюсь iPhone 6s — всё летает.',

'У меня последний самсунг, всё хорошо.',

'Зачем платить больше, если есть Xiaomi?',

'Лучший телефон - Nokia 3310. У меня до сих пор работает.',

'Только последний iPhone — там улучшили водонепроницаемость.'

];

Чтобы лучше понимать аудиторию, вы хотите отобрать всех пользователей, которые пользуются iPhone. Применим для этого регулярное выражение.

Скопировать кодJAVASCRIPT

const regex = /iPhone/;

const iPhoneComments = comments.filter((item) => regex.test(item));

*/\* метод test проверяет, есть ли искомая строка*

*в тексте и возвращает true или false.*

*Позже разберём этот метод подробно. \*/*

console.log(iPhoneComments);

*/\* [*

*"Я пользуюсь iPhone 6s — всё летает.",*

*"Только последний iPhone — там улучшили водонепроницаемость."*

*] \*/*

Из пяти человек двое — пользователи айфона. Мы узнали это благодаря регулярным выражениям. Такой поиск в строке можно выполнить методом includes — результат был бы тем же. Но это самый простой пример. Обычно регулярными выражениями решают сложные задачи.

Таким образом, регулярные выражения — способ поиска строк в тексте. В этом уроке наш поиск был незатейлив: мы искали в строке слово. Чтобы настроить более сложный поиск, нужно разобраться со спецсимволами, флагами и методами регулярных выражений. Обо всём этом речь пойдёт дальше.

Пример

Исправьте регулярное выражение regex так, чтобы найти в строке цифры из названия модели Порше.

*const car = 'Porsche 911 Carrera';*

*const regex = /911/; // исправьте регулярное выражение*

*const carModel = car.match(regex);*

*console.log(carModel);*

# Методы и флаги

Регулярные выражения позволяют выполнять очень сложные алгоритмы поиска. Чтобы уметь эти алгоритмы составлять, нужно изучить синтаксис регулярных выражений — так же, как вы учили синтаксис JavaScript.

На поиск влияют три сущности:

* Спецсимволы — содержатся в самóм регулярном выражении и позволяют тоньше задать поиск. Например, искать в тексте сразу «Яндекс» и «Yandex».
* Методы — определяют, что нужно делать, когда строка найдена: заменить её или сообщить, где произошло совпадение.
* Флаги — кастомизируют работу метода (увеличивают возможности). Например, выставив флаг, можно найти все совпадения с регуляркой или только одно.

Таким образом: спецсимволы определяют, что искать; методы — что делать; а флаги — как делать.

## Методы работы с регулярными выражениями

Все методы для работы с регулярными выражениями ищут текст в строке, а различаются только назначением.

Есть тонкость. Методы для работы с регулярными выражениями бывают двух типов:

1. Методы строк. Их вызывают как метод строки:

Скопировать кодJAVASCRIPT

const regex = /л/;

const word = 'солнце';

word.match(regex); *// [ "л" ] — метод нашёл символ в строке*

*/\* match — строковый метод. Поэтому мы вызываем его*

*как метод строки, а регулярное выражение передаём как аргумент. \*/*

1. Методы регулярных выражений. Их вызывают как метод регулярки:

Скопировать кодJAVASCRIPT

const regex = /т/;

const word = 'лестница';

regex.test(word); *// true — метод подтвердил, что в строке есть совпадение*

*/\* test — метод объекта RegExp. Поэтому мы вызываем его*

*как метод переменной regExp и передаём строку как аргумент. \*/*

Запутаться тут очень легко. Поэтому будем писать имена методов вместе с объектом, в который они записаны: String.match и RegExp.test.

Пойдём от простого к сложному: рассмотрим один метод и на нём покажем работу всего остального.

### Метод String.match

Этот метод мы применяли в прошлом уроке. String.match нужен для поиска совпадений: им вы можете проверить, есть ли слово в тексте, и посчитать, сколько раз оно встречается.

Метод String.match принимает один аргумент — регулярное выражение. Если найти строку удалось, String.match вернёт массив с тем, что нашёл:

Скопировать кодJAVASCRIPT

const regex = /сс/;

const word = 'рассчитывать';

word.match(regex); *// [ "сс" ]*

Если символ найти не удалось, String.match вернёт null. Обратите внимание: не пустой массив, а именно null.

Скопировать кодJAVASCRIPT

const regex = /сс/;

const word = 'расчёт';

word.match(regex); *// null*

Метод String.match можно настроить: например, заставить его считать все совпадения в тексте, а не только первое. Для такой настройки и нужны флаги.

## Флаги

Флаг — символ, который стоит в регулярном выражении в самом конце (после слеша) и задаёт настройки поиска. Всего их шесть: g, i, m, u, y, s. В регулярном выражении флагов может быть несколько, и они могут идти в любом порядке.

Регулярное выражение с флагом выглядит так:

Скопировать кодJAVASCRIPT

const regex1 = /смысл жизни/g; *// здесь мы установили флаг g*

const regex2 = /соль/gi; *// тут установлены флаги g и i*

Из них важно знать три: g, i, m. В этом уроке мы разберём их.

### Флаг g

Если установить флаг g, мы будем искать все совпадения, а не только первое:

Скопировать кодJAVASCRIPT

const regex = /с/;

const regexGlobal = /с/g;

const word = 'искусство';

word.match(regex); *// [ "с" ]*

word.match(regexGlobal); *// [ "с", "с", "с" ];*

### Флаг i

Если поставить флаг i, поиск не будет различать строчных и прописных букв:

Скопировать кодJAVASCRIPT

const str = 'Вильгельм Конрад Рентген стал лауреатом Нобелевской премии в 1901 году.'

const regex = /рентген/;

const regexIgnore = /рентген/i;

str.match(regex); *// null*

str.match(regexIgnore); *// [ "Рентген" ]*

### Флаг m

Флаг m сработает, только если мы используем определённые спецсимволы, которых вы пока не знаете. Мы сначала разберём эти спецсимволы, а потом вернёмся к m.

### Флаг u

Оставшимися тремя флагами u, y и s пользуются довольно редко.

Если поставить флаг u, можно искать символы по их номеру в Юникоде.

Чтобы найти символ по коду, используют такую конструкцию: \u{номер символа}.

Например:

Скопировать кодJAVASCRIPT

const str = '& — это амперсанд. Его номер в таблице Юникод: 26';

const regex = /\u{26}/; *// здесь флаг u не установлен*

const regexUnicode = /\u{26}/u; *// а здесь — установлен*

str.match(regex); *// null*

str.match(regexUnicode); *// ["&"]*

### Флаг y

Флаг y позволяет проверить, есть ли искомая строка в строго определённом месте текста. Работает так: вы включаете флаг y при составлении регулярного выражения. А затем записываете в свойство lastIndex этого регулярного выражения номер символа, с которого нужно проверять:

Скопировать кодJAVASCRIPT

const fuelUp = 'заправляемся, потом платим';

let regex = /платим/y;

regex.lastIndex = 0;

fuelUp.match(regex);

*// null. Первое слово «заправляемся», а не «платим»*

regex.lastIndex = 20;

fuelUp.match(regex); *// [ "платим" ]*

*/\* Интересно, что в разных браузерах информация*

*строковых методов выводится в разном объёме:*

*где только индекс, а где также исходная строка и скобочные группы \*/*

## Коротко о методах и флагах

Ваши инструменты для работы с регулярными выражениями — это спецсимволы, методы и флаги. Спецсимволы настраивают поиск, методы определяют, что делать, когда строка найдена. Флаги же кастомизируют поиск.

В этом уроке вы познакомились с методом String.match — он ищет строку и возвращает массив с найденными значениями. Как именно он будет искать, определяют флаги. То есть флаги влияют на поведение всего шаблона.

Основных флага три: g, i и m. Флаг g настраивает глобальный поиск: будет найдено не только первое совпадение с регулярным выражением, а вообще все совпадения.

Флаг i позволяет искать слова без учёта регистра. Если этот флаг включён, поиск не будет различать слова «Гриб» и «гриб».

О третьем флаге m расскажем позже. Выполняйте задания и переходите к следующему уроку. Там подробно поговорим о спецсимволах.

# Спецсимволы и их обратные классы

В прошлом уроке мы говорили о флагах — одной из трёх сущностей для работы с регулярными выражениями. Теперь переходим к спецсимволам. Расскажем об основных, а ещё разберём, что такое обратные классы спецсимволов. Поехали.

Представьте, что к нашей статье о телефонах появились новые комментарии. Теперь из всех комментаторов мы хотим отобрать пользователей тех моделей, у которых есть s на конце: Xs, 5s, 6s. Напишем регулярку:

Скопировать кодJAVASCRIPT

const regex = /iPhone .s/gi; *// Поставим точку вместо номера модели*

Нам приходит на помощь точка. В шаблоне регулярного выражения она заменяет любой символ. Неважно, какая цифра стоит перед s, поэтому на её место в шаблоне мы ставим точку.

В шаблоне может быть сколько угодно точек.

Скопировать кодJAVASCRIPT

const str = `

Не помню, как именно его зовут:

то ли Сортини, то ли Сордини.

Быть может, и Сардини, через «а».

`;

const regex = /С.р.ини/g; *// Точки в регулярном выражении заменяют любой символ.*

str.match(regex); *// [ "Сортини", "Сордини", "Сардини" ]*

Но у точки есть одна слабость: она не найдёт перенос строки.

Скопировать кодJAVASCRIPT

const str = `Внезапный

перенос`;

const regex = /Внезапный.перенос/;

str.match(regex); *// null*

Но не беда — эту проблему можно решить спецсимволами.

## Спецсимволы

Спецсимволы — это такие конструкции из символов в регулярных выражениях, которые позволяют искать не один символ, а сразу группу.

Есть одиночные спецсимволы — точка, например. Но многие спецсимволы — это буквы. И чтобы движок понимал, что это не простая буква, а особый символ, в регулярном выражении такой символ нужно экранировать — то есть поставить перед ним обратный слеш \w.

Если экранировать спецсимвол, который должен идти без слеша, движок будет его воспринимать как обычный символ. Например, если поставить обратный слеш перед точкой, движок будет искать только точки, а не любые символы:

Скопировать кодJAVASCRIPT

const str1 = 'yandex.ru/maps/';

const regex1 = /\.ru/; *// экранировали точку*

const regex2 = /\/maps/; *// экранировали слеш перед словом maps*

str1.match(regex1); *// [ ".ru" ]*

str1.match(regex2); *// [ "/maps" ]*

*// Чтобы найти обратный слеш, его тоже нужно экранировать*

const str2 = 'C:\\';

const regex3 = /\\/; *// экранировали слеш*

str2.match(regex3); *// [ "\" ]*

### Спецсимвол \w - цифру, лат. Букву, подчёркивание

Этот спецсимвол говорит движку искать любую цифру, латинскую букву или нижнее подчёркивание:

Скопировать кодJAVASCRIPT

const str = 'Простите, я отправил старую версию файла диплом\_финал\_2\_копия\_3.docx. Не смотрите не неё. Высылаю вам актуальную диплом\_финал\_2\_копия\_4.docx';

const regex = /диплом\wфинал\w\w\wкопия\w\w.docx/g;

str.match(regex);

*// [ "диплом\_финал\_2\_копия\_3.docx", "диплом\_финал\_2\_копия\_4.docx" ]*

У спецсимвола \w есть обратный класс: \W. Если \w ищет цифры, латинские буквы и нижние подчёркивания, то \W ищет всё что угодно: знаки вопроса, пробелы, слеши, кириллические символы, — но только не цифры, буквы базовой латиницы и нижние подчёркивания.

Итак, обратный класс — это всё равно, что сказать «ищи НЕ такой-то спецсимвол».

Скопировать кодJAVASCRIPT

const str = `

Даты основания некоторых IT-компаний:

Yandex: 23.09.1997

Apple: 01/04/1976

IBM: 16-06-1911

`;

const regex = /\w\w\W\w\w\W\w\w\w\w/g;

*/\* цифры в дате мы обозначили строчной \w, а разделители —*

*заглавной \W. Разделитель это НЕ цифра, НЕ латинская буква*

*и НЕ нижнее подчёркивание. \*/*

str.match(regex); *// [ "23.09.1997", "01/04/1976", "16-06-1911" ]*

### Спецсимвол \d совпадает с любой цифрой

Спецсимвол \d совпадает с любой цифрой:

Скопировать кодJAVASCRIPT

const str = 'Владивосток 2000';

const regex = /\d\d\d\d/g;

str.match(regex); *// [ "2000" ]*

Обратный класс этого символа совпадает со всеми не-цифрами: буквами, пробелами, спецсимволами.

Скопировать кодJAVASCRIPT

*// найдёт все символы, кроме цифр*

const someSymbol = /\D/g;

const string = 'I was born in 1987';

string.match(someSymbol);

*// ["I", " ", "w", "a", "s", " ", "b", "o", "r", "n", " ", "i", "n", " "]*

### Спецсимвол \s пробелы, переносы строк и табуляции

Спецсимвол \s ищет «пустоты» в тексте: пробелы (в том числе неразрывные), переносы строк и табуляции.

Скопировать кодJAVASCRIPT

const str = 'Сдают паспорта,\n' +

' и я' +

' сдаю';

const regex = /\s/g;

str.match(regex).length; *// 32 — Маяковский любил пробелы*

Его обратный класс \S ищет любые символы, только не пробелы, табуляции и переносы:

Скопировать кодJAVASCRIPT

const str = 'Сдают паспорта,\n' +

' и я' +

' сдаю';

const regex = /\S/g;

str.match(regex).length; *// 22*

*// букв с запятыми получилось на десяток меньше*

### Ещё немного о методе String.match

Мы говорили, что он возвращает массив найденных значений. И у этого массива есть важная особенность.

Если передать методу String.match регулярное выражение без флага g, полученному массиву будут определены дополнительные свойства:

Скопировать кодJAVASCRIPT

const str = 'тро-ло-ло';

const result = str.match(/ло/);

result[0]; *// "ло"*

result.index; *// 4*

result.input; *// "тро-ло-ло"*

Если искать по регулярному выражению с флагом g, дополнительных свойств в итоговом массиве не будет. Мы получим просто массив, состоящий из строк:

Скопировать кодJAVASCRIPT

const str = 'тро-ло-ло';

const result = str.match(/ло/g);

result; *// ["ло", "ло"]*

### Спецсимвол \b означает границу слова

Возвращаемся к спецсимволам. \b означает границу слова. Движок считает границей слова:

* начало строки;
* конец строки;
* любой символ, кроме цифр, латинских букв и нижнего подчёркивания.

Так мы можем находить символы в начале и в конце строки:



Скопировать кодJAVASCRIPT

const string = 'sadness';

string.match(/\bs/).index; *// 0 — это первая буква s*

*// спецсимвол указал границу левее неё, т. е. начало*

string.match(/s\b/).index; *// 6 - а это последняя*

*// спецсимвол указал границу правее неё, т. е. конец*

Обратите внимание: любая нелатинская буква прочитывается движком как отдельное слово. Поэтому с кириллицей спецсимвол \b не работает.

Скопировать кодJAVASCRIPT

*/\* Попробуем найти в номере автомобиля код региона.*

*Поскольку он в конце строки и состоит из цифр,*

*составим такое регулярное выражение: \*/*

'А333НР99'.match(/\d\d\b/);

*// ["33", index: 2, input: "А333НР99", groups: undefined]*

*/\* Мы хотели получить регион, но получили цифры*

*в середине номера, потому что движок счёл кириллический*

*символ «Н» за конец слова «333». \*/*

Но на самом деле в автомобильных номерах допускаются только буквы, подобные которым есть в латинице. Если записать номер латинскими буквами, всё получается:

Скопировать кодJAVASCRIPT

'A333HP99'.match(/\d\d\b/);

*//["99", index: 6, input: "A333HP99", groups: undefined]*

Также есть обратный класс \B — он ищет не конец слова: цифру, латинскую букву или нижнее подчёркивание. Таким образом, \B делает то же самое, что \w.

Полный перечень спецсимволов [можно посмотреть на MDN](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/RegExp).

# Наборы и диапазоны

В прошлом уроке мы говорили о спецсимволах и обратных классах. Они позволяют искать совпадения не с конкретным символом, а с одним из группы. Например, \d ищет совпадения с любой цифрой.

В этом уроке расскажем, как такие группы символов можно настраивать самостоятельно. Например, искать совпадения только с цифрами от 3 до 8. Инструменты, которыми мы будем эту настройку проводить, называются наборы и диапазоны.

## Наборы

Наборы позволяют задать группу символов и искать совпадения с одним из них. Например, искать сразу и букву «а», и «б»:

Скопировать кодJAVASCRIPT

*// Ищем все символы «а» и «б»:*

'барабан'.match(/[аб]/g); *// [ "б", "а", "а", "б", "а" ]*

Чтобы создать набор, нужно перечислить подходящие символы в квадратных скобках:

Скопировать кодJAVASCRIPT

const str = '14/03/2018';

const regex = /\d\d\W0[345]\W2018/g; *// этот шаблон ищет все дни весны*

str.match(regex);

## Диапазоны

Диапазоны позволяют указать промежуток доступных значений. Например, искать только цифры от 5 до 8 или только буквы от «М» до «Р».

Скопировать кодJAVASCRIPT

const regex = /[м-р]/gi;

'Марсианин'.match(regex); *// [ "М", "р", "н", "н" ]*

Диапазон — часть набора. Это два символа, разделённых дефисом.

С числами, например:

Скопировать кодJAVASCRIPT

const str = '20/04/2019';

const regex = /\d\d\W0[1-6]\W2019/g; *// этот шаблон ищет все дни первого полугодия*

str.match(regex); *// [ "20/04/2019" ]*

Диапазоны можно совмещать с наборами и другими диапазонами:

Скопировать кодJAVASCRIPT

const regex = /[a-z0-9\-]/gi; *// все латинские буквы, все цифры и дефис*

const str = 'В Берлин я летал на boeing 737-800';

str.match(regex).join(''); *// "boeing737-800"*

## Кириллица

В прошлом уроке говорили о спецсимволе \w — он ищет все латинские символы, цифры и нижние подчёркивания. Но с текстами, написанными кириллическими символами, он нам никак не помогает. Зато помогают наборы и диапазоны.

Скопировать кодJAVASCRIPT

const regex = /[\wа-я\s]/gi;

*/\* Ищем всю латиницу и всю кириллицу, все пробелы,*

*а от знаков препинания избавляемся. \*/*

const str = 'Yandex 2020. Ещё не вечер!';

str.match(regex).join(''); *// "Yandex 2020 Ещ не вечер"*

Обратите внимание: буква «ё» исчезла. Дело в том, что в таблице символов она стоит обособленно от других кириллических букв. Поэтому, чтобы не терять «ё», дописывайте её в набор:

Скопировать кодJAVASCRIPT

const regex = /[\wа-я\sё]/gi;

*/\* Ищем всю латиницу и всю кириллицу,*

*а от знаков препинания избавляемся. \*/*

const str = 'Yandex 2020. Ещё не вечер!';

str.match(regex).join(''); *// "Yandex 2020 Ещё не вечер"*

*// Теперь «ё» не потерялась!*

## Исключающие наборы и диапазоны

Мы рассказывали, что у спецсимволов есть обратные классы — они ищут все те символы, которые их антагонист игнорирует.

У наборов и диапазонов тоже есть противоположности — исключающие наборы и диапазоны. Они ищут все символы, кроме тех, что мы им перечислим.

Чтобы сделать набор или диапазон исключающим, перед ним нужно поставить «шляпку» ^ (карет):

Скопировать кодJAVASCRIPT

const str = 'Оценки за четверть: 2 3 3 5 2 4 2 3 5';

const regex = /[^1-3]/g;

str.match(regex).join(''); *// "Оценки за четверть: 5 4 5"*

*// картина стала получше, но большие пустоты выдают обман*

Обратите внимание: если дефис просто стоит в регулярном выражении, движок не будет его искать. Сам по себе дефис спецсимволом не является.

Однако, если он стоит в наборе между двумя символами, движок начнёт воспринимать это не как набор трёх символов, а как диапазон:

Скопировать кодJAVASCRIPT

/[ac-]/; *// это набор из трёх символов: a, c и -*

/a-c/; *// это уже диапазон из трёх символов: a, b и с*

Чтобы набор непредвиденно не превращался в диапазон, дефис нужно экранировать:

Скопировать кодJAVASCRIPT

/a\-c/; *// теперь это набор из a, c и -*

Мы рекомендуем всегда экранировать дефис в регулярных выражениях: так можно избежать ошибок. То же самое касается вот этих символов:

* дефис -,
* точка .,
* плюс +,
* круглые скобки (),
* карет (шляпка) ^,
* открывающая квадратная скобка [.

Их можно не экранировать в регулярном выражении. Но всё же мы рекомендуем это делать, чтобы не было путаницы.

## Иногда точка — это просто точка

Теперь у вас есть новые инструменты для работы с регулярными выражениями — наборы и диапазоны. Они позволяют задавать группы символов и искать совпадения только с символами из этих групп. Также можно написать исключающий набор или диапазон, чтобы намеренно не искать определённые символы.

Помните, что буква «ё» не входит в диапазон кириллических букв от «а» до «я», поэтому её следует отдельно добавлять в набор вот так: /а-яё/i.

Регулярные выражения становятся всё сложнее. Но всё ещё обладают одним значительным недостатком — они фиксированной длины. Пока что мы не можем искать слова с разным количеством букв. Решить эту проблему позволяют квантификаторы. О них мы расскажем в следующем уроке.

# Квантификаторы

В прошлом уроке мы говорили о наборах и диапазонах. В этом — поговорим о квантификаторах. Они позволяют искать любое количество повторов. А ещё расскажем, почему квантификаторам не чужды человеческие пороки, и что нам это даёт.

## Основные квантификаторы

Предположим, мы хотим найти слово «свиной», но не помним, сколько в нём букв «н»:

Скопировать кодJAVASCRIPT

const str = 'Сколько букв «н» в слове «свиной»?';

Для такого поиска нам нужно регулярное выражение. Оно ищет строку, которая начинается на «сви», затем идёт одна или две буквы «н», и конец слова: «ой».

### квантификатор + от одного до бесконечности повторений

Решить эту задачу поможет квантификатор +. Если поставить плюс после символа, движок ищет все слова, где этот символ встречается один раз или больше:

Скопировать кодJAVASCRIPT

const str = 'Правильно писать «свиной», с одной «н»';

const regex = /свин+ой/;

*// такое регулярное выражение найдёт оба варианта: и с «н», и с «нн»*

str.match(regex); *// [ "свиной" ]*

### квантификатор \* от нуля до бесконечности повторений

Другой квантификатор для поиска повторяющихся символов — это звёздочка (астериск): \*. Работает почти как плюс, но есть одно отличие. Символ, после которого стоит плюс, обязательно должен быть в строке. То есть плюс как бы говорит движку: «Ищи вот этот символ и после него ещё может быть сколько угодно таких же символов».

Звёздочку же устраивает и полное отсутствие того символа, который перед ней стоит. То есть если звёздочка стоит после символа, этого символа в найденной строке может вовсе не быть:

Скопировать кодJAVASCRIPT

const exc = 'экскаватор';

const esc = 'эскалатор';

const regex = /эк\*ска[вл]атор/; *// буква "к" может встречаться, а может и не встречаться*

exc.match(regex); *// [ "экскаватор" ]*

esc.match(regex); *// [ "эскалатор" ]*

### квантификатор ? необязательный символ

Ещё один способ сделать символ необязательным: квантификатор ?. Если звёздочка приемлет от нуля до бесконечности символов, то знак вопроса строже — его устраивает лишь 0 и 1: либо есть один символ, либо его нет вообще.

Скопировать кодJAVASCRIPT

*/\* делает букву u необязательной и будет искать*

*оба варинта написания favourite и favorite. \*/*

const regex = /favou?rite/g;

const str = 'favourite for favorite';

str.match(regex); *// ["favourite", "favorite"]*

### квантификатор {} контроль повторов

Для поиска набора повторяющихся символов мы перечисляли их подряд в регулярном выражении:

Скопировать кодJAVASCRIPT

const regionCode = /\d\d\d/;

const phoneNumber = 'Мой номер телефона: +7(999)123-12-21';

phoneNumber.match(regionCode); *// [ "999" ]*

Квантификатор {} позволяет этого избежать. После символа можно поставить фигурные скобки и указать в них, сколько повторений этого символа мы хотим найти:

Скопировать кодJAVASCRIPT

const regionCode = /\d{3}/;

const phoneNumber = 'Мой номер телефона: +7(999)123-12-21';

phoneNumber.match(regionCode); *// [ "999" ]*

Можно также указывать не точное количество повторений, а диапазон. Например, когда нас устроит от 2 до 5 повторений:

Скопировать кодJAVASCRIPT

const str = 'столько, стооолько и вот стооооооооолько';

const regex = /сто{2,5}лько/;

str.match(regex); *// [ "стооолько" ]*

*// в слове "стооооооооолько" повторов "о" больше 5*

Верхнюю границу числа повторений можно не указывать: например, можно искать от 5 до бесконечности. В таких случаях просто не пишите в фигурных скобках второе число. Но запятую ставьте — она нужна.

Скопировать кодJAVASCRIPT

const someSymbol = /a{1,}/g;

const str = 'alohaa';

console.log(str.match(someSymbol)); *// ["a", "aa"]*

## Шпаргалка

| **ИМЯ** | **ИЩЕТ** |
| --- | --- |
| + | от одного до бесконечности совпадений с символом подряд. |
| \* | от нуля до бесконечности совпадений с символом подряд. |
| ? | от нуля до одного совпадений с символом. |
| {} | точное количество совпадений или диапазон совпадений с символом подряд. |

# Начало и конец строки. Флаг m

Мы говорили, что основные флаги — это g, i и m. Первые два мы объяснили, а об m обещали рассказать позже. В этом уроке поговорим о спецсимволах, обозначающих начало и конец строки. А это как раз позволит разобраться с флагом m.

## Начало строки и конец строки

Начнём с начала. Спецсимвол, обозначающий начало строки, — шляпка ^, или карет. Он говорит шаблону, что нужно искать совпадение только с началом текста:

Скопировать кодJAVASCRIPT

const regex = /^\d+/g;

const newReg = /\d+/g;

const str = '2001 год: Космическая одиссея, вышел в 1968 году';

str.match(regex); *// [ "2001" ];*

str.match(newReg); *// [ "2001", "1968" ];*

Символ конца строки — знак доллара $. Аналогично указывает, что нужно искать совпадение с концом текста:

Скопировать кодJAVASCRIPT

const regex = /\d+$/;

const str = 'https://praktikum.yandex.ru/trainer/frontend-developer/lesson/45';

str.match(regex); *// ( ["45"] )*

## Многострочные тексты

Чтобы написать многострочный текст, перенос можно поставить комбинацией символов \n или пользоваться обычными переносами в шаблонных строках:

Скопировать кодJAVASCRIPT

const str1 = `Председатель Совнаркома, Наркомпроса, Мининдела!

Эта местность мне знакома, как окраина Китая!`;

const str2 = 'Это утро, радость эта,\nЭта мощь и дня и света,'

Тут есть расхождение между логикой компьютера и человека. Для нас перенос означает, что строка закончилась. А для движка всё иначе — он считает началом строки только то, что стоит после открывающей кавычки, а концом — до закрывающей:

Скопировать кодJAVASCRIPT

const str = `Председатель Совнаркома, Наркомпроса, Мининдела!

Эта местность мне знакома, как окраина Китая!`;

const regex = /[А-Я]\*!$/gi; *// найдём последние слова каждой строки*

str.match(regex); *// ["Китая!"] — получилось только последнее слово всего текста*

Чтобы изменить такое поведение движка, нужно установить флаг m. Тогда движок будет считать каждый перенос концом одной строки и началом другой:

Скопировать кодJAVASCRIPT

const str = `Эта личность мне знакома! Знак допроса вместо тела.

Многоточие шинели. Вместо мозга — запятая.`;

const regex = /[А-Я]\*\.$/gim;

str.match(regex); *// [ "тела.", "запятая." ]*

Это работает и с шаблонными строками, и с обычными: в них движок будет считать началом и концом строки комбинации \n:

Скопировать кодJAVASCRIPT

const regex = /^I got/gm;

const str = 'I got, I got, I got, I got\n' +

'Loyalty, got royalty\n' +

'Inside my DNA\n' +

'Coconut quarter piece, got war and peace\n' +

'Inside my DNA\n' +

'I got power, poison, pain and joy\n' +

'Inside my DNA\n' +

'I got hustle, though, ambition, flow\n' +

'Inside my DNA'

str.match(regex); *// [ "I got", "I got", "I got" ]*

## Трудности с точкой

В многострочных текстах точка работает плохо, потому что не учитывает конца строки. Попробуем найти самую длинную строку (считаем с конца), которая состоит из любых символов:

Скопировать кодJAVASCRIPT

const regex = /.\*$/;

const regexMultiline = /.\*$/m;

const str = 'I got, I got, I got, I got\n' +

'Loyalty, got royalty\n' +

'Inside my DNA\n' +

'Coconut quarter piece, got war and peace\n' +

'Inside my DNA\n' +

'I got power, poison, pain and joy\n' +

'Inside my DNA\n' +

'I got hustle, though, ambition, flow\n' +

'Inside my DNA'

str.match(regexMultiline); *// [ "I got, I got, I got, I got" ]*

*/\* Флаг m включён, поэтому движок отсчитал*

*сколько мог любых символов от конца первой строки \*/*

str.match(regex); *// [ "Inside my DNA" ]*

*/\* Тут флаг m выключен, поэтому движок отсчитывает*

*символы от конца строки. Точка в шаблоне означает*

*любой символ, кроме переноса. Поэтому мы нашли*

*только последнюю строчку \*/*

При работе с многострочным текстом точка иногда подводит — она означает любой символ, кроме конца строки. Поэтому, если нужно искать совпадение с абсолютно любым символом, пользуйтесь набором [\s\S]:

Скопировать кодJAVASCRIPT

const regex = /[\s\S]\*$/; *// вариант с началом строки: /^[\s\S]\*/;*

const str = 'I got, I got, I got, I got\n' +

'Loyalty, got royalty\n' +

'Inside my DNA\n' +

'Coconut quarter piece, got war and peace\n' +

'Inside my DNA\n' +

'I got power, poison, pain and joy\n' +

'Inside my DNA\n' +

'I got hustle, though, ambition, flow\n' +

'Inside my DNA'

*// Теперь мы получим весь текст целиком*

str.match(regex); *// [ "I got, I got, I got, I got\nLoyalty, got royalty\nInside my DNA\nCoconut quarter piece, got war and peace\nInside my DNA\nI got power, poison, pain and joy\nInside my DNA\nI got hustle, though, ambition, flow\nInside my DNA" ]*

Таким образом, начало строки обозначают шляпкой ^, а конец — знаком доллара $. По умолчанию весь текст — одна строка, у которой есть один конец и одно начало.

Такое поведение движка можно изменить, поставив флаг m. Флаг m говорит движку, чтобы тот воспринимал каждый перенос как конец одной строки и начало другой.

Пример

Составьте список рифм этого стихотворения. Нужно вывести последнее слово каждой строки.

*const regex = /[а-яё]+\S\*$/gim; // напишите здесь регулярное выражение с шаблоном и флагами*

*const str = `То отголосок юных дней*

*В душе проснулся, замирая,*

*И в блеске утренних лучей,*

*Казалось, ночь была немая.*

*То сон предутренний сошёл,*

*И дух, на грани пробужденья,*

*Воспрянул, вскрикнул и обрёл*

*Давно мелькнувшее виденье.*

*То был безжалостный порыв*

*Бессмертных мыслей вне сомнений.*

*И он умчался, пробудив*

*Толпы забытых откровений.*

*То бесконечность пронесла*

*Над падшим духом ураганы.*

*То Вечно-Юная прошла*

*В неозаренные туманы.`;*

*console.log(str.match(regex)); // ["дней", "замирая,", "лучей,", "немая.", ...]*

# Методы для регулярных выражений

Хорошие новости: мы закончили со спецсимволами и флагами. Осталось изучить несколько удобных методов. Вы уже знакомы с методом String.match. В этом уроке расскажем ещё о четырёх.

## Метод String.search

Метод String.search принимает на вход регулярное выражение и возвращает индекс символа, на котором произошло совпадение:

Скопировать кодJAVASCRIPT

const regex = /\d{3,}/i;

const string = '12! равен 479001600';

string.search(regex); *// 10*

Метод String.search не работает с глобальным поиском — он всегда возвращает индекс первого совпадения:

Скопировать кодJAVASCRIPT

const regex = /\d/ig;

const string = '12! равен 479001600';

string.search(regex); *// 0 - индекс первого совпадения, то есть "1".*

Если же совпадения не было вовсе — нам вернётся -1:

Скопировать кодJAVASCRIPT

const regex = /[тут]$/i;

const string = "Тут царь мимо не проходил?";

string.search(regex); *// -1*

## Метод String.split

Метод String.split вы уже встречали в теме о примитивах. Им мы создавали из строки массив:

Скопировать кодJAVASCRIPT

'Пришёл. Увидел. Победил.'.split(' '); *// ["Пришёл.", "Увидел.", "Победил."]*

'Пришёл. Увидел. Победил.'.split('. '); *// ["Пришёл", "Увидел", "Победил."]*

Тут мы указываем разделитель как строку. Методу String.split можно передать на вход и регулярное выражение. Тогда на выходе мы получим массив элементов, которые находили между строчками, найденными нашей регуляркой:

Скопировать кодJAVASCRIPT

const regex = /\n/im;

`Мой дядя самых честных правил,

Когда не в шутку занемог,

Он уважать себя заставил

И лучше выдумать не мог.`.split(regex);

*/\* [*

*"Мой дядя самых честных правил,",*

*"Когда не в шутку занемог,",*

*"Он уважать себя заставил",*

*"И лучше выдумать не мог."*

*] \*/*

Обратите внимание: метод String.split ожидает, что какой-то элемент должен быть и до, и после разделителя. Поэтому если ваша строка начинается с разделителя или заканчивается разделителем, в полученном массиве первым или последним элементом будет пустая строка.

## Метод RegExp.exec

Метод RegExp.exec похож на String.match. Если в регулярном выражении нет флага g, String.match и RegExp.exec вернут одинаковый результат:

Скопировать кодJAVASCRIPT

let regex = /\w+/;

let str = 'Someone must have been telling lies about Josef K.';

str.match(regex); *// ["Someone"]*

regex.exec(str); *// ["Someone"]*

Но если установить флаг g, метод RegExp.exec будет вести себя совсем иначе. Он вернёт первое совпадение, а затем — запишет в свойство lastIndex регулярного выражения номер символа в тексте, на котором это совпадение произошло.

Если запустить метод RegExp.exec снова, он будет искать не с начала строки, а с того символа, на котором остановился в прошлый раз. То есть с записанного в свойство lastIndex:

Скопировать кодJAVASCRIPT

const str = `Идёт без проволочек

И тает ночь, пока

Над спящим миром лётчик

Уходит в облака.`;

let regex = /.+/g;

regex.exec(str); *// ["Идёт без проволочек"]*

regex.exec(str); *// ["И тает ночь, пока"]*

regex.exec(str); *// ["Над спящим миром лётчик"]*

regex.exec(str); *// ["Уходит в облака."]*

RegExp.exec обновляет свойство lastIndex после каждого вызова.

Таким образом, метод RegExp.exec помнит, где закончил поиски в прошлый раз. Если запустить этот метод в цикле, мы можем точно контролировать, сколько совпадений хотим найти.

Если очередной поиск не дал результатов, в свойство lastIndex регулярного выражения записывается 0. То есть новый поиск начнётся с начала.

## Метод RegExp.test

Мы уже упоминали о методе RegExp.test. Он вызывается на регулярном выражении, а строку принимает на вход. Результат работы метода — булево значение. Если совпадение найти удалось, RegExp.test вернёт true:

Скопировать кодJAVASCRIPT

const regex = /\w+@\w+\.\w+/; *// шаблон для поиска email*

const str = 'Stas Basov: stasbasov@yandex.ru';

regex.test(str); *// true*

Если же совпадений с регулярным выражением в строке нет, получим false:

Скопировать кодJAVASCRIPT

const regex = /\w+@\w+\.\w+/;

const str = 'Stas Basov';

regex.test(str); *// false - в строке нет email.*

Метод RegExp.test тоже обладает памятью — как и RegExp.exec. Если передать ему регулярное выражение с флагом g, в свойство lastIndex этого регулярного выражения попадёт номер символа, где было найдено совпадение. И следующий поиск регулярного выражения начнётся с этого символа, а не с начала строки:

Скопировать кодJAVASCRIPT

const regex = /\w+@\w+\.\w+/g;

const str = 'Stas Basov: stas-basov@yandex.ru';

regex.test(str); *// true*

regex.lastIndex; *// 32*

*// Вызовем метод RegExp.test ещё раз:*

regex.test(str); *// false*

regex.lastIndex; *// 0*

*/\* предыдущий поиск не дал результатов,*

*поэтому свойство lastIndex было сброшено до нуля \*/*

## Метод String.replace

И, наконец, последний метод — String.replace. Он позволяет на ходу собирать новый текст из исходного. Работает он так: ищет в тексте совпадение с регуляркой, составляет какую-то новую строку из полученной и возвращает её нам.

Всего у метода два параметра. Первый — это шаблон регулярного выражения. Вторым аргументом может быть строка или функция. Именно второй параметр определяет, как именно будет составлена новая строка из совпадения.



Чаще всего методу String.replace передают строку. Все совпадения в исходном тексте будут заменены этой строкой.

Скопировать кодJAVASCRIPT

const strObj = 'Пробел всегда нужно ставить после запятой ,а не до неё.';

const regex = /\s,/g;

strObj.replace(regex, ', '); *// "Пробел всегда нужно ставить после запятой, а не до неё."*

В этой строке можно использовать спецсимволы. Тогда движок не будет просто заменять совпадения какой-то строкой. Спецсимволы применяются редко, поэтому мы не будем разбирать их в этой теме.

# Регулярные выражения. Заключение

Мы рассказали вам многое о регулярных выражениях. Уложить всё в голове непросто. Это нормально — даже опытные программисты часто лезут в интернет, чтобы вспомнить, как работает какой-нибудь спецсимвол. Советуем выполнять задания — так вы быстрее набьёте руку.

Вот несколько ссылок на полезные материалы и инструменты. Они помогут писать сложные регулярки:

* [на этом сайте можно визуализировать алгоритм](https://bit.ly/3m7axbc), который задаёт регулярное выражение для поиска;
* [тут можно проверить, что регулярка ищет в тексте](https://regex101.com/);
* [статья о валидации форм регулярными выражениями](https://css-tricks.com/form-validation-part-1-constraint-validation-html/).