Регулярные выражения в Javascript

Оглавление

[Синтаксис языка 0](#_Toc444164282)

[Внешние скрипты и порядок их исполнения 1](#_Toc444164283)

[Структура кода 2](#_Toc444164284)

[Комментарии 2](#_Toc444164285)

[Переменные 3](#_Toc444164286)

[Константы 3](#_Toc444164287)

[Типы данных 4](#_Toc444164288)

[Оператор typeof 5](#_Toc444164289)

[Современный стандарт, "use strict" 6](#_Toc444164290)

[Домашнее задание 6](#_Toc444164291)

[Используемая литература 7](#_Toc444164292)

# Методы RegExp и String

Регулярные выражения в JavaScript являются объектами класса RegExp.

Кроме того, методы для поиска по регулярным выражениям встроены прямо в обычные строки String.

К сожалению, общая структура встроенных методов слегка запутана, поэтому мы сначала рассмотрим их по отдельности, а затем – рецепты по решению стандартных задач с ними.

**str.search(reg)**

Метод возвращает позицию первого совпадения или -1, если ничего не найдено.

var str = "Люблю регэкспы я, но странною любовью";

alert( str.search( /лю/i ) ); // 0

Ограничение метода search – он всегда ищет только первое совпадение.

Нельзя заставить search искать дальше первого совпадения, такой синтаксис попросту не предусмотрен. Но есть другие методы, которые это умеют.

**str.match(reg) без флага g**

Метод str.match работает по-разному, в зависимости от наличия или отсутствия флага g, поэтому сначала мы разберём вариант, когда его нет.

В этом случае str.match(reg) находит только одно, первое совпадение.

Результат вызова – это массив, состоящий из этого совпадения, с дополнительными свойствами index – позиция, на которой оно обнаружено и input – строка, в которой был поиск.

Например:

var str = "ОЙ-Ой-ой";

var result = str.match( /ой/i );

alert( result[0] ); // ОЙ (совпадение)

alert( result.index ); // 0 (позиция)

alert( result.input ); // ОЙ-Ой-ой (вся поисковая строка)

У этого массива не всегда только один элемент.

Если часть шаблона обозначена скобками, то она станет отдельным элементом массива.

Например:

var str = "javascript - это такой язык";

var result = str.match( /JAVA(SCRIPT)/i );

alert( result[0] ); // javascript (всё совпадение полностью)

alert( result[1] ); // script (часть совпадения, соответствующая скобкам)

alert( result.index ); // 0

alert( result.input ); // javascript - это такой язык

Благодаря флагу i поиск не обращает внимание на регистр буквы, поэтому находит javascript. При этом часть строки, соответствующая SCRIPT, выделена в отдельный элемент массива.

Позже мы ещё вернёмся к скобочным выражениям, они особенно удобны для поиска с заменой.

str.match(reg) с флагом g

При наличии флага g, вызов match возвращает обычный массив из всех совпадений.

Никаких дополнительных свойств у массива в этом случае нет, скобки дополнительных элементов не порождают.

Например:

var str = "ОЙ-Ой-ой";

var result = str.match( /ой/ig );

alert( result ); // ОЙ, Ой, ой

Пример со скобками:

var str = "javascript - это такой язык";

var result = str.match( /JAVA(SCRIPT)/gi );

alert( result[0] ); // javascript

alert( result.length ); // 1

alert( result.index ); // undefined

Из последнего примера видно, что элемент в массиве ровно один, и свойства index также нет. Такова особенность глобального поиска при помощи match – он просто возвращает все совпадения.

Для расширенного глобального поиска, который позволит получить все позиции и, при желании, скобки, нужно использовать метод RegExp#exec, которые будет рассмотрен далее.

**str.split(reg|substr, limit)**

Разбивает строку в массив по разделителю – регулярному выражению regexp или подстроке substr.

Обычно мы используем метод split со строками, вот так:

alert('12-34-56'.split('-')) // [12, 34, 56]

Можно передать в него и регулярное выражение, тогда он разобьёт строку по всем совпадениям.

Тот же пример с регэкспом:

alert('12-34-56'.split(/-/)) // [12, 34, 56]

**str.replace(reg, str|func)**

Швейцарский нож для работы со строками, поиска и замены любого уровня сложности.

Его простейшее применение – поиск и замена подстроки в строке, вот так:

// заменить дефис на двоеточие

alert('12-34-56'.replace("-", ":")) // 12:34-56

При вызове со строкой замены replace всегда заменяет только первое совпадение.

Чтобы заменить все совпадения, нужно использовать для поиска не строку "-", а регулярное выражение /-/g, причём обязательно с флагом g:

// заменить дефис на двоеточие

alert( '12-34-56'.replace( /-/g, ":" ) ) // 12:34:56

В строке для замены можно использовать специальные символы:

Спецсимволы Действие в строке замены

* $$ Вставляет "$".
* $& Вставляет всё найденное совпадение.
* $` Вставляет часть строки до совпадения.
* $' Вставляет часть строки после совпадения.
* $\*n\* где n -- цифра или двузначное число, обозначает n-ю по счёту скобку, если считать слева-направо.

Пример использования скобок и $1, $2:

var str = "Василий Пупкин";

alert(str.replace(/(Василий) (Пупкин)/, '$2, $1')) // Пупкин, Василий

Ещё пример, с использованием $&:

var str = "Василий Пупкин";

alert(str.replace(/Василий Пупкин/, 'Великий $&!')) // Великий Василий Пупкин!

Для ситуаций, которые требуют максимально «умной» замены, в качестве второго аргумента предусмотрена функция.

Она будет вызвана для каждого совпадения, и её результат будет вставлен как замена.

Например:

var i = 0;

// заменить каждое вхождение "ой" на результат вызова функции

alert("ОЙ-Ой-ой".replace(/ой/gi, function() {

return ++i;

})); // 1-2-3

В примере выше функция просто возвращала числа по очереди, но обычно она основывается на поисковых данных.

Эта функция получает следующие аргументы:

* str – найденное совпадение,
* p1, p2, ..., pn – содержимое скобок (если есть),
* offset – позиция, на которой найдено совпадение,
* s – исходная строка.

Если скобок в регулярном выражении нет, то у функции всегда будет ровно 3 аргумента:replacer(str, offset, s).

Используем это, чтобы вывести полную информацию о совпадениях:

// вывести и заменить все совпадения

function replacer(str, offset, s) {

alert( "Найдено: " + str + " на позиции: " + offset + " в строке: " + s );

return str.toLowerCase();

}

var result = "ОЙ-Ой-ой".replace(/ой/gi, replacer);

alert( 'Результат: ' + result ); // Результат: ой-ой-ой

С двумя скобочными выражениями – аргументов уже 5:

function replacer(str, name, surname, offset, s) {

return surname + ", " + name;

}

var str = "Василий Пупкин";

alert(str.replace(/(Василий) (Пупкин)/, replacer)) // Пупкин, Василий

Функция – это самый мощный инструмент для замены, какой только может быть. Она владеет всей информацией о совпадении и имеет доступ к замыканию, поэтому может всё.

[regexp.test(str)](http://learn.javascript.ru/regexp-methods" \l "regexp-test-str)

Теперь переходим к методам класса RegExp.

Метод test проверяет, есть ли хоть одно совпадение в строке str. Возвращает true/false.

Работает, по сути, так же, как и проверка str.search(reg) != -1, например:

var str = "Люблю регэкспы я, но странною любовью";

// эти две проверки идентичны

alert( /лю/i.test(str) ) // true

alert( str.search(/лю/i) != -1 ) // true

Пример с отрицательным результатом:

var str = "Ой, цветёт калина...";

alert( /javascript/i.test(str) ) // false

alert( str.search(/javascript/i) != -1 ) // false

**[regexp.exec(str)](http://learn.javascript.ru/regexp-methods" \l "regexp-exec-str)**

Для поиска мы уже видели методы:

* search – ищет индекс
* match – если регэксп без флага g – ищет совпадение с подрезультатами в скобках
* match – если регэксп с флагом g – ищет все совпадения, но без скобочных групп.

Метод regexp.exec дополняет их. Он позволяет искать и все совпадения и скобочные группы в них.

Он ведёт себя по-разному, в зависимости от того, есть ли у регэкспа флаг g.

* Если флага g нет, то regexp.exec(str) ищет и возвращает первое совпадение, является полным аналогом вызова str.match(reg).
* Если флаг g есть, то вызов regexp.exec возвращает первое совпадение и запоминает его позицию в свойстве regexp.lastIndex. Последующий поиск он начнёт уже с этой позиции. Если совпадений не найдено, то сбрасывает regexp.lastIndex в ноль.

Это используют для поиска всех совпадений в цикле:

var str = 'Многое по JavaScript можно найти на сайте http://javascript.ru';

var regexp = /javascript/ig;

var result;

alert( "Начальное значение lastIndex: " + regexp.lastIndex );

while (result = regexp.exec(str)) {

alert( 'Найдено: ' + result[0] + ' на позиции:' + result.index );

alert( 'Свойство lastIndex: ' + regexp.lastIndex );

}

alert( 'Конечное значение lastIndex: ' + regexp.lastIndex );

Здесь цикл продолжается до тех пор, пока regexp.exec не вернёт null, что означает «совпадений больше нет».

Найденные результаты последовательно помещаются в result, причём находятся там в том же формате, что и match – с учётом скобок, со свойствами result.index и result.input.

# Классы и спецсимволы

Рассмотрим практическую задачу – есть телефонный номер "+7(903)-123-45-67", и нам нужно найти в этой строке цифры. А остальные символы нас не интересуют.

Для поиска символов определённого вида в регулярных выражениях предусмотрены «классы символов».

Класс символов – это специальное обозначение, под которое подходит любой символ из определённого набора.

Например, есть класс «любая цифра». Он обозначается \d. Это обозначение вставляется в шаблон, и при поиске под него подходит любая цифра.

То есть, регулярное выражение /\d/ ищет ровно одну цифру:

var str = "+7(903)-123-45-67";

var reg = /\d/;

// не глобальный регэксп, поэтому ищет только первую цифру

alert( str.match(reg) ); // 7

…Ну а для поиска всех цифр достаточно добавить к регэкспу флаг g:

var str = "+7(903)-123-45-67";

var reg = /\d/g;

alert( str.match(reg) ); // массив всех совпадений: 7,9,0,3,1,2,3,4,5,6,7

**Важнейшие классы: \d \s \w**

Это был класс для цифр.

Конечно же, есть и другие.

Наиболее часто используются:

* \d (от английского «digit» – «цифра»)
  + Цифра, символ от 0 до 9.
* \s (от английского «space» – «пробел»)
  + Пробельный символ, включая табы, переводы строки и т.п.
* \w (от английского «word» – «слово»)

Символ «слова», а точнее – буква латинского алфавита или цифра или подчёркивание '\_'. Не-английские буквы не являются \w, то есть русская буква не подходит.

Например, \d\s\w обозначает цифру, за которой идёт пробельный символ, а затем символ слова.

Регулярное выражение может содержать одновременно и обычные символы и классы.

Например, CSS\d найдёт строку CSS, с любой цифрой после неё:

var str = "Стандарт CSS4 - это здорово";

var reg = /CSS\d/

alert( str.match(reg) ); // CSS4

И много классов подряд:

alert( "Я люблю HTML5!".match(/\s\w\w\w\w\d/) ); // 'HTML5'

Совпадение (каждому классу в регэкспе соответствует один символ результата):

http://learn.javascript.ru/article/regexp-character-classes/love-html5-classes.png

**Граница слова \b**

Граница слова \b – это особый класс.

Он интересен тем, что обозначает не символ, а границу между символами.

Например, \bJava\b найдёт слово Java в строке Hello, Java!, но не в строке Hello, Javascript!.

alert( "Hello, Java!".match(/\bJava\b/) ); // Java

alert( "Hello, Javascript!".match(/\bJava\b/) ); // null

Граница имеет «нулевую ширину» в том смысле, что обычно символам регулярного выражения соответствуют символы строки, но не в этом случае.

Граница – это проверка.

При поиске движок регулярных выражений идёт по шаблону и одновременно по строке, пытаясь построить соответствие. Когда он видит \b, то проверяет, что текущая позиция в строке подходит под одно из условий:

* Начало текста, если первый символ \w.
* Конец текста, если последний символ \w.
* Внутри текста, если с одной стороны \w, а с другой – не \w.

Например, в строке Hello, Java! под \b подходят следующие позиции:

http://learn.javascript.ru/article/regexp-character-classes/hello-java-boundaries.png

Как правило, \b используется, чтобы искать отдельно стоящее слово. Не на русском конечно, хотя подобную проверку, как мы увидим далее, можно легко сделать для любого языка. А вот на английском, как в примере выше или для чисел, которые являются частным случаем \w – легко.

Например, регэксп \b\d\d\b ищет отдельно двузначные числа. Иными словами, он требует, чтобы до и после \d\d был символ, отличный от \w (или начало/конец текста).

Обратные классы

Для каждого класса существует «обратный ему», представленный такой же, но заглавной буквой.

«Обратный» – означает, что ему соответствуют все остальные символы, например:

* \D Не-цифра, то есть любой символ кроме \d, например буква.
* \S Не-пробел, то есть любой символ кроме \s, например буква.
* \W Любой символ, кроме \w, то есть не латинница, не подчёркивание, не цифра. В частности, русские буквы принадлежат этому классу.
* \B Проверка, обратная \b.

В начале этой главы мы видели, как получить из телефона +7(903)-123-45-67 все цифры.

Первый способ – найти все цифры через match(/\d/g).

Обратные классы помогут реализовать альтернативный – найти все НЕцифры и удалить их из строки:

var str = "+7(903)-123-45-67";

alert( str.replace(/\D/g, "") ); // 79031234567

Пробелы – обычные символы

Заметим, что в регулярных выражениях пробел – такой же символ, как и другие.

Обычно мы не обращаем внимание на пробелы. Для нашего взгляда строки 1-5 и 1 - 5 почти идентичны.

Однако, если регэксп не учитывает пробелов, то он не сработает.

Попытаемся найти цифры, разделённые дефисом:

alert( "1 - 5".match(/\d-\d/) ); // null, нет совпадений!

Поправим это, добавив в регэксп пробелы:

alert( "1 - 5".match(/\d - \d/) ); // работает, пробелы вокруг дефиса

Конечно же, пробелы в регэкспе нужны лишь тогда, когда мы их ищем. Лишние пробелы (как и любые лишние символы) могут навредить:

alert( "1-5".match(/\d - \d/) ); // null, так как в строке 1-5 нет пробелов

Короче говоря, в регулярном выражении все символы имеют значение. Даже (и тем более) – пробелы.

Точка – любой символ

Особым классом символов является точка ".".

В регулярном выражении, точка "." обозначает любой символ, кроме перевода строки:

alert( "Z".match(/./) ); // найдено Z

Посередине регулярного выражения:

var re = /CS.4/;

alert( "CSS4".match(re) ); // найдено "CSS4"

alert( "CS-4".match(re) ); // найдено "CS-4"

alert( "CS 4".match(re) ); // найдено "CS 4" (пробел тоже символ)

Обратим внимание – точка означает именно «произвольный символ».

То есть какой-то символ на этом месте в строке должен быть:

alert( "CS4".match(/CS.4/) ); // нет совпадений, так как для точки нет символа

**Экранирование специальных символов**

В регулярных выражениях есть и другие символы, имеющие особый смысл.

Они используются, чтобы расширить возможности поиска.

Вот их полный список: [ \ ^ $ . | ? \* + ( ).

Не пытайтесь запомнить его – когда мы разберёмся с каждым из них по отдельности, он запомнится сам собой.

Чтобы использовать специальный символ в качестве обычного, он должен бытьэкранирован.

Или, другими словами, перед символом должен быть обратный слэш '\'.

Например, нам нужно найти точку '.'. В регулярном выражении она означает «любой символ, кроме новой строки», поэтому чтобы найти именно сам символ «точка» – её нужно экранировать: \..

alert( "Глава 5.1".match(/\d\.\d/) ); // 5.1

Круглые скобки также являются специальными символами, так что для поиска именно скобки нужно использовать \(. Пример ниже ищет строку "g()":

alert( "function g()".match(/g\(\)/) ); // "g()"

Сам символ слэш '/', хотя и не является специальными символом в регулярных выражениях, но открывает-закрывает регэксп в синтаксисе /...pattern.../, поэтому его тоже нужно экранировать.

Так выглядит поиск слэша '/':

alert( "/".match(/\//) ); // '/'

Ну и, наконец, если нам нужно найти сам обратный слэш \, то его нужно просто задублировать.

Так выглядит поиск обратного слэша "\":

alert( "1\2".match(/\\/) ); // '\'

# Наборы и диапазоны

Например, [еао] означает любой символ из этих трёх: 'а', 'е', или 'о'.

Такое обозначение называют набором. Наборы используются в регулярном выражении наравне с обычными символами:

// найти [г или т], а затем "оп"

alert( "Гоп-стоп".match(/[гт]оп/gi) ); // "Гоп", "топ"

Обратим внимание: несмотря на то, что в наборе указано несколько символов, в совпадении должен присутствовать ровно один из них.

Поэтому в примере ниже нет результатов:

// найти "В", затем [у или а], затем "ля"

alert( "Вуаля".match(/В[уа]ля/) ); // совпадений нет

Поиск подразумевает:

* В,
* затем одну из букв набора [уа],
* а затем ля

Таким образом, совпадение было бы для строки Вуля или Валя.

**Диапазоны**

Квадратные скобки могут также содержать диапазоны символов.

Например, [a-z] – произвольный символ от a до z, [0-5] – цифра от 0 до 5.

В примере ниже мы будем искать "x", после которого идёт два раза любая цифра или буква от A до F:

// найдёт "xAF"

alert( "Exception 0xAF".match(/x[0-9A-F][0-9A-F]/g) );

Обратим внимание, в слове Exception есть сочетание xce, но оно не подошло, потому что буквы в нём маленькие, а в диапазоне [0-9A-F] – большие.

Если хочется искать и его тоже, можно добавить в скобки диапазон a-f: [0-9A-Fa-f]. Или же просто указать у всего регулярного выражения флаг i.

Символьные классы – всего лишь более короткие записи для диапазонов, в частности:

* \d – то же самое, что [0-9],
* \w – то же самое, что [a-zA-Z0-9\_],
* \s – то же самое, что [\t\n\v\f\r ] плюс несколько юникодных пробельных символов.

В квадратных скобках можно использовать и диапазоны и символьные классы – вместе.

Например, нам нужно найти все слова в тексте. Если они на английском – это достаточно просто:

var str = "The sun is rising!";

alert( str.match(/\w+/g) ); // The, sun, is, rising

В этом примере мы забежали немного вперёд и использовали pattern\w+, что означает один или более символов, подходящих под классpattern\w. Позже мы рассмотрим + детальнее, а пока – давайте посмотрим, найдутся ли слова на русском?

var str = "Солнце встаёт!";

alert( str.match(/\w+/g) ); // null

Ничего не найдено! Это можно понять, ведь \w – это именно английская букво-цифра, как можно видеть из аналога [a-zA-Z0-9\_].

Чтобы находило слово на русском – нужно использовать диапазон, например /[а-я]/.

А чтобы на обоих языках – и то и другое вместе:

var str = "Солнце (the sun) встаёт!";

alert( str.match(/[\wа-я]+/gi) ); // Солнце, the, sun, вста, т

…Присмотритесь внимательно к предыдущему примеру! Вы видите странность? Оно не находит букву ё, более того – считает её разрывом в слове. Причина – в кодировке юникод, она подробно раскрыта в главе [Строки](http://learn.javascript.ru/string).

Буква ё лежит в стороне от основной кириллицы и её следует добавить в диапазон дополнительно, вот так:

var str = "Солнце (the sun) встаёт!";

alert( str.match(/[\wа-яё]+/gi) ); // Солнце, the, sun, встаёт

Теперь всё в порядке.

**Диапазоны «кроме»**

Кроме обычных, существуют также исключающие диапазоны: [^…].

Квадратные скобки, начинающиеся со знака каретки: [^…] находят любой символ, кроме указанных.

Например:

* [^аеуо] – любой символ, кроме 'a', 'e', 'y', 'o'.
* [^0-9] – любой символ, кроме цифры, то же что \D.
* [^\s] – любой не-пробельный символ, то же что \S.

Пример ниже ищет любые символы, кроме букв, цифр и пробелов:

alert( "alice15@gmail.com".match(/[^\d\sA-Z]/gi) ); // "@", "."

**Не нужно экранирование**

Обычно, если мы хотим искать именно точку, а не любой символ, или именно символ \, то мы используем экранирование: указываем \. или \\.

В квадратных скобках большинство специальных символов можно использовать без экранирования, если конечно они не имеют какой-то особый смысл именно внутри квадратных скобок.

То есть, «как есть», без экранирования можно использовать символы:

* Точка '.'.
* Плюс '+'.
* Круглые скобки '( )'.
* Дефис '-', если он находится в начале или конце квадратных скобок, то есть не выделяет диапазон.
* Символ каретки '^', если не находится в начале квадратных скобок.
* А также открывающая квадратная скобка '['.

То есть, точка "." в квадратных скобках означает не «любой символ», а обычную точку.

Регэксп [.,] ищет один из символов «точка» или «запятая».

В примере ниже регэксп [-().^+] ищет один из символов -().^. Они не экранированы:

// Без экранирования

var re = /[-().^+]/g;

alert( "1 + 2 - 3".match(re) ); // найдёт +, -

…Впрочем, даже если вы решите «на всякий случай» заэкранировать эти символы, поставив перед ними обратный слэш \ – вреда не будет:

// Всё заэкранировали

var re = /[\-\(\)\.\^\+]/g;

alert( "1 + 2 - 3".match(re) ); // тоже работает: +, -

# Квантификаторы +, \*, ? и {n}

Рассмотрим ту же задачу, что и ранее – взять телефон вида +7(903)-123-45-67 и найти все числа в нём. Но теперь нас интересуют не цифры по отдельности, а именно числа, то есть результат вида 7, 903, 123, 45, 67.

Для поиска цифр по отдельности нам было достаточно класса \d. Но здесь нужно искать числа – последовательности из 1 или более цифр.

Количество {n}

Количество повторений символа можно указать с помощью числа в фигурных скобках: {n}.

Такое указание называют квантификатором (от англ. quantifier).

У него есть несколько подформ записи:

Точное количество: {5}

Регэксп \d{5} обозначает ровно 5 цифр, в точности как \d\d\d\d\d.

Следующий пример находит пятизначное число.

alert( "Мне 12345 лет".match(/\d{5}/) ); // "12345"

Количество от-до: {3,5}

Для того, чтобы найти, например, числа размером от трёх до пяти знаков, нужно указать границы в фигурных скобках: \d{3,5}

alert( "Мне не 12, а 1234 года".match(/\d{3,5}/) ); // "1234"

Последнее значение можно и не указывать. Тогда выражение \d{3,} найдет числа, длиной от трех цифр:

alert( "Мне не 12, а 345678 лет".match(/\d{3,}/) ); // "345678"

В случае с телефоном нам нужны числа – одна или более цифр подряд. Этой задаче соответствует регулярное выражение \d{1,}:

var str = "+7(903)-123-45-67";

alert( str.match(/\d{1,}/g) ); // 7,903,123,45,67

Короткие обозначения

Для самых часто востребованных квантификаторов есть специальные короткие обозначения.

+

Означает «один или более», то же что {1,}.

Например, \d+ находит числа – последовательности из 1 или более цифр:

var str = "+7(903)-123-45-67";

alert( str.match(/\d+/g) ); // 7,903,123,45,67

?

Означает «ноль или один», то же что и {0,1}. По сути, делает символ необязательным.

Например, регэксп ou?r найдёт o, после которого, возможно, следует u, а затем r.

Этот регэксп найдёт or в слове color и our в colour:

var str = "Можно писать color или colour (британский вариант)";

alert( str.match(/colou?r/g) ); // color, colour

\*

Означает «ноль или более», то же что {0,}. То есть, символ может повторяться много раз или вообще отсутствовать.

Пример ниже находит цифру, после которой идёт один или более нулей:

alert( "100 10 1".match(/\d0\*/g) ); // 100, 10, 1

Сравните это с '+' (один или более):

alert( "100 10 1".match(/\d0+/g) ); // 100, 10

Ещё примеры

Эти квантификаторы принадлежат к числу самых важных «строительных блоков» для сложных регулярных выражений, поэтому мы рассмотрим ещё примеры.

Регэксп «десятичная дробь» (число с точкой внутри): \d+\.\d+

В действии:

alert( "0 1 12.345 7890".match(/\d+\.\d+/g) ); // 12.345

Регэксп «открывающий HTML-тег без атрибутов», такой как <span> или <p>: /<[a-z]+>/i

Пример:

alert( "<BODY> ... </BODY>".match(/<[a-z]+>/gi) ); // <BODY>

Это регулярное выражение ищет символ '<', за которым идут одна или более букв английского алфавита, и затем '>'.

Регэксп «открывающий HTML-тег без атрибутов» (лучше): /<[a-z][a-z0-9]\*>/i

Здесь регулярное выражение расширено: в соответствие со стандартом, HTML-тег может иметь символ цифры на любой позиции, кроме первой, например <h1>.

alert( "<h1>Привет!</h1>".match(/<[a-z][a-z0-9]\*>/gi) ); // <h1>

Регэксп «открывающий или закрывающий HTML-тег без атрибутов»: /<\/?[a-z][a-z0-9]\*>/i

В предыдущий паттерн добавили необязательный слэш /? перед тегом. Его понадобилось заэкранировать, чтобы JavaScript не принял его за конец шаблона.

alert( "<h1>Привет!</h1>".match(/<\/?[a-z][a-z0-9]\*>/gi) ); // <h1>, </h1>

# Домашнее задание

* Напишите регулярное выражение для поиска HTML-цвета, заданного как #ABCDEF, то есть # и содержит затем 6 шестнадцатеричных символов.
* Создайте регэксп, который ищет все положительные числа, в том числе и с десятичной точкой. Например, var str = "1.5 0 12. 123.4."
* Время может быть в формате часы:минуты или часы-минуты. И часы и минуты состоят из двух цифр, например 09:00, 21-30. Напишите регулярное выражение для поиска времени
* Написать проверку правильности координаты в файле инициализации шахматной доски c помощью регулярного выражения. Он должен иметь формат вида A6.
* Написать регулярные выражения для следующих сущностей: номер телефона в формате +7(965)-123-45-67, email, серии и номера паспорта. Применить написанные регулярные выражения необходимо для валидации произвольной формы, в которой обязательно должны присутствовать описанные выше поля. Поля, которые проходят валидацию подсветить зеленым, остальные – красным.

# Используемая литература

* [Mozilla Developer Network](https://developer.mozilla.org/)
* [MSDN](http://msdn.microsoft.com/)
* [Safari Developer Library](https://developer.apple.com/library/safari/navigation/index.html)
* [Современный учебник JavaScript](http://learn.javascript.ru)