***Регулярные выражения*** *- это такие* ***команды*** *для сложного* ***поиска и замены*** *(или просто поиска). Они позволяют делать очень интересные вещи, но, к сожалению, довольно тяжелы в освоении.*

*Существует несколько методов JavaScript для работы с регулярными выражениями. Мы начнем знакомится с ними на примере replace - вы уже знакомы с этим методом: он первым параметром принимает что менять, а вторым - на что менять. А сам метод применяется к строке, в которой производится замена.*

**Метод replace**

Первым параметром этого метода можно передавать не просто строку, а **регулярное выражение**. Посмотрите разницу:

где\_меняем.replace(что меняем, на что меняем)

alert( 'bab'.replace('a', '!') ); //выведет 'b!b'

alert( 'bab'.replace(/a/g, '!') ); //выведет 'b!b'

Обратите внимание на слеши **/**, в которых стоит буква 'a'. Эти слеши называются *ограничителями* регулярных выражений.

Ограничители нужны для того, чтобы после них писать *модификаторы* - команды, которые изменяют общие свойства регулярного выражения.

С одним модификатором - **g** - вы должны быть уже знакомы - мы немного касались его, когда проходили метод **replace**. Этот модификатор включает режим глобального поиска и замены - без него регулярка ищет только первое совпадение, а с ним - все совпадения. Посмотрите разницу:

alert( 'aab'.replace(/A/g, '!') ); //выведет '!ab'

alert( 'aab'.replace(/A/g', '**!**') ); //выведет '!!b'

Есть также модификатор **i** - он заставляет регулярки **игнорировать регистр** символов:

alert( 'aAb'.replace(/A/g', '**!**') ); //выведет 'a!b'

alert( 'aAb'.replace(/A/gi', '**!**') ); //выведет '!!b'

//Во втором случае игнорируется регистр символов.

В регулярке можно указывать несколько модификаторов, их порядок значения не имеет.

**Буквы, цифры, любой символ**

В регулярных выражениях есть два типа символов: те, которые обозначают **сами себя** и **символы-команды** (специальные символы).

**Буквы и цифры** обозначают сами себя, а вот **точка** является специальным символом и обозначает *'любой символ'*. Смотрите примеры:

'xax xaax'.replace(/xax/g, '!'); //вернет '! xaax'

'123 xaax'.replace(/123/g, '!'); //вернет '! xaax'

'x3x xaax'.replace(/x3x/g, '!'); //вернет '! xaax'

//Обратите внимание на то, что регистр имеет значение:

'a3b A3B'.replace(/A3B/g, '!'); //вернет 'a3b !'

Далее посмотрите примеры с использованием спецсимвола 'точка':

'xax xsx x&x x-x xaax'.replace(/x.x/g, '!'); //вернет '! ! ! ! xaax'

Так как **точка** - это **любой символ**, то под нашу регулярку попадут все подстроки по такому шаблону: **буква 'x', затем любой символ, затем опять буква 'x'**. Первые 4 подстроки попали под этот шаблон (это **xax xsx x&x x-x**) и заменились на '!', а последняя подстрока (**xaax**) не попадает, так как внутри (между буквами 'x') не один символ - а целых два.

'xax xabx'.replace(/x..x/g, '!'); //вернет 'xax !'

Так как **точка** - это **любой символ**, а в нашей регулярке идут **две точки подряд**, то под нашу регулярку попадут все подстроки по такому шаблону: **буква 'x', затем два любых символа, затем опять буква 'x'**. Первая подстрока не попала под этот шаблон (так как у нее только 1 символ между буквами 'x'), а последняя подстрока (**xabx**) - попала.

Итак, запомните: буквы и цифры обозначают сами себя, а точка заменяет любой символ.

**Операторы повторения символов (\*,+,?)**

Бывают ситуации, когда мы хотим указать, что **символ повторяется** заданное количество раз. Если мы знаем точное число повторений, то можно просто написать его несколько раз - (/aaaa/). Но что делать, если мы хотим сказать такое: '*повторить один или более раз*'?

Для этого существуют **операторы** *(квантификаторы)* **повторения**: плюс **'+'** (один и более раз), звездочка **'\*'** (ноль и более раз) и вопрос **'?'** (ноль или один раз, иначе говоря - может быть, а может не быть).

Данные операторы действуют на тот символ, который стоит **перед ними**.

Посмотрите примеры:

'xx xax xaax xaaax xbx'.replace(/xa+x/g, '!'); //вернет 'xx ! ! ! xbx'

В данном случае шаблон поиска выглядит так: *буква 'x', буква 'a' один или более раз, буква 'x'*.

'xx xax xaax xaaax xbx'.replace(/xa\*x/g, '!'); //вернет '! ! ! ! xbx'

В данном случае шаблон поиска выглядит так: *буква 'x', буква 'a' ноль или более раз, буква 'x'*. По-другому можно сказать так: буквы 'a' или нет, или повторяется один или более раз.

Кроме очевидного варианта **xax xaax xaaax**, под шаблон также попадает подстрока **'xx'**, так как там нет буквы **'a'** вообще (то есть 0 раз).

А **'xbx'** не попал, так как там нет буквы **'a'**, но есть буква **'b'** (ее мы не разрешили).

'xx xax xaax xbx'.replace(/xa?x/g, '!'); //вернет '! ! xaax xbx'

В данном случае шаблон поиска выглядит так: *буква 'x', далее буква 'a' может быть или не быть, потом буква 'x'*.

**Группирующие скобки**

В предыдущих примерах операторы повторения действовали только на один символ, который стоял перед ними. Что делать, если мы хотим подействовать им на несколько символов?

Для этого существуют группирующие скобки **'(' и ')'**:

'xabx xababx xaabbx'.replace(/x(ab)+x/g, '!'); //вернет '! ! xaabbx'

В данном случае шаблон поиска выглядит так: *буква 'x', далее строка 'ab' один или более раз, потом буква 'x'*.

То есть: если что-то стоит в группирующих скобках и сразу после ')' стоит оператор повторения - он подействует на все, что стоит внутри скобок.

**Экранировка спецсимволов**

Предположим, что мы хотим сделать так, чтобы спецсимвол обозначал сам себя. Например, для того, чтобы найти по такому шаблону: *буква 'a', затем плюс '+', затем буква 'x'*. Следующий код будет работать не так, как хотелось бы:

'a+x ax aax aaax'.replace(/a+x/g, '!'); //вернет 'a+x ! ! !''

Автор регулярки хотел, чтобы шаблон поиска выглядел так: *буква 'a', затем плюс '+', затем буква 'x'*.

А на самом деле он выглядит так: *буква 'a' один или более раз, потом буква 'x'*.

Поэтому подстрока **'a+x'** и не попала под шаблон (мешает '+') - а все остальные попали.

Итак, правило: чтобы спецсимвол обозначал сам себя - его нужно экранировать с помощью **обратного слеша**. Вот так:

'a+x ax aax aaax'.replace(/a\+x/g, '!'); //вернет 'a+x ! ! !'

Вот теперь шаблон поиска выглядит так, как надо: *буква 'a', затем плюс '+', затем буква 'x'*.

'a.x abx azx'.replace(/a\.x/g, '!'); //вернет '! abx azx'

В данном примере шаблон выглядит так: *буква 'a', затем точка '.', затем буква 'x'*. Сравните со следующим примером (забыт обратный слеш):

'a.x abx azx'.replace(/a.x/g, '!'); //вернет '! ! !'

Все подстроки попали под шаблон, так как **незаэкранированная** точка обозначает любой символ.

Обратите внимание на то, что если вы забудете обратный слеш для точки (когда она должна обозначать сама себя) - этого можно даже не заметить:

'a.x'.replace(/a.x/g, '!'); //вернет '!', как мы и хотели

Визуально работает правильно (так как точка обозначает любой символ, в том числе и обычную точку '.'). Но если поменять строку, в которой происходят замены - мы увидим нашу ошибку:

'a.x abx azx'.replace(/a.x/g, '!'); //вернет '! ! !', а ожидалось '! abx azx'

Будьте внимательны!

**Список специальных символов и обычных**

Если экранировать обычный символ - ничего страшного не случится - он все равно будет обозначать сам себя.

**Исключение** - цифры, они станут карманами.

Часто возникает сомнение, является ли данный символ специальным. Некоторые доходят до того, что экранируют все подозрительные символы подряд. Однако, это **плохая практика** (захламляет регулярку обратными слешами).

Являются спецсимволами: **$ ^ . \* + ? \ {} [] () |**

Не являются спецсимволами: **@ : , ' '' ; - \_ = < > % # ~ `& ! /**

**Ограничение жадности**

Чтобы понять, о чем пойдет речь - посмотрите пример:

//Выведет '! e', а ожидалось '! qw x e':

'a23e4x qw x e'.replace(/a.+x/g, '!');

В данном примере шаблон поиска выглядит так: **буква 'a', затем любой символ один или более раз, затем буква 'x'**.

Однако, регулярка сработала не так, как ожидал автор - она захватила **максимально возможное** количество символов, то есть закончилась не на первом иксе **'x'**, а на последнем **'x'**.

Такое поведение операторов (квантификаторов) повторения называется *жадностью* - они стремятся забрать как можно больше.

Такая особенность не всегда полезна и, к счастью, ее можно отменить, говоря по-другому - *ограничить жадность*.

Делается это с помощью добавления знака **'?'** к оператору повторения: вместо жадных **'+'** и **'\*'** мы напишем **'+?'** и **'\*?'** - они будут не такие жадные:

'a23e4x qw x e'.replace(/a.+?x/g, '!'); вернет '! qw x e'

В данном примере шаблон поиска выглядит так: **буква 'a', затем любой символ один или более раз (с ограничением жадности), затем буква 'x'**.

С помощью **'?'** мы ограничили жадность плюсу - и теперь он ищет **до первого** совпадения.

Жадность можно ограничивать всем операторам повторения, в том числе и **'?'**, и **'{}'** - вот так: **'??'** и **'{}?'**.

## Фигурные скобки

Операторы '+', '\*', '?' не слишком универсальные (хотя и используются очень часто). А если нам нужно указать конкретное число повторений?

На помощь придет оператор **{}**, указывающий количество повторений.

Работает он следующим образом: **{5}** – пять повторений, **{2,5}** – повторяется от двух до пяти (оба включительно), **{2,}** - повторяется два и более раз. Обратите внимание на то, что такого варианта - **{,2}** - нет. Посмотрите примеры:

'xx xax xaax xaaax'.replace(/xa{1,2}x/g, '!'); //вернет 'xx ! ! xaaax'

В данном случае шаблон поиска выглядит так: *буква 'x', буква 'a' один или два раза, буква 'x'*.

'xx xax xaax xaaax'.replace(/xa{2,}x/g, '!'); //вернет 'xx xax ! !'

В данном случае шаблон поиска выглядит так: *буква 'x', буква 'a' два раза и более, буква 'x'*.

'xx xax xaax xaaax'.replace(/xa{2}x/g, '!'); //вернет 'xx xax ! xaaax'

В данном случае шаблон поиска выглядит так: *буква 'x', буква 'a' два раза, буква 'x'*.

Данную задачу, конечно же, можно решить и так (здесь это будет даже короче):

'xx xax xaax xaaax'.replace(/xaax/g, '!'); //вернет 'xx xax ! xaaax'

Однако второй вариант не всегда короче - сравните два выражения:

'строка'.replace(/xaaaaaaaaaax/g, '!');

'строка'.replace(/xa{10}x/g, '!');

Согласитесь - теперь второе проще.

А вот так работать не будет:

'xax xaax xaaax'.replace(/xa{,2}x/g, '!'); //ожидали '! ! xaaax'

Мы хотели такой шаблон: *буква 'x', буква 'a' два раза и меньше, буква 'x'*, но, к сожалению, такое не работает.

Нужно указать явно:

'xx xax xaax xaaax'.replace(/xa{1,2}x/g, '!'); //вернет '! ! xaaax'

Вот теперь получим то, что хотели.

Обратите внимание на то, что **ноль тоже допустим**:

'xx xax xaax xaaax'.replace(/xa{0,2}x/g, '!'); //вернет '! ! ! xaaax'

В данном случае шаблон поиска выглядит так: *буква 'x', буква 'a' ноль, один или два раза, буква 'x'*.

Под шаблон попала и подстрока **'xx'** - там буква 'a' стоит ноль раз, то есть ее нет совсем.

## Группы символов \s, \S, \w, \W, \d, \D

Существуют специальные команды, которые позволяют выбрать сразу целые группы символов. Например, **\d** означает 'цифра от 0 до 9', а **\D** – наоборот, 'не цифра'. Посмотрите примеры:

'1 12 123 abc @@@'.replace(/\d+/g, '!'); //вернет '! ! ! abc @@@'

В данном примере шаблон поиска выглядит так: *цифра от 0 до 9 один или более раз*.

Обратите внимание на то, что оператор **+** считает данные команды **одним целым** (группирующие скобки не нужны). Это относится ко всем операторам повторения.

'123abc3@@'.replace(/\D+/g, '!'); //вернет '123!3!'

А теперь все будет наоборот: *все что угодно, но не цифра от 0 до 9, один или более раз*.

### Что обозначают разные группы символов:

|  |  |
| --- | --- |
| **Символ** | **Его значение** |
| \s | Обозначает пробел или пробельный символ (имеется ввиду перевод строки, табуляция и т.п.). |
| \S | Не пробел, то есть всё, кроме \s. |
| \w | Цифра или буква (кириллица не входит в буквы). |
| \W | Не цифра и не буква. |
| \d | Цифра от 0 до 9. |
| \D | Не цифра от 0 до 9. |

Как вы можете заметить, большая буква означает отрицание: если **\s** - пробел, то **\S** – не пробел и так далее. Посмотрите примеры для закрепления:

'1 12 123 abc @@@'.replace(/\s/g, '!'); //вернет '1!12!123!abc!@@@'

В данном примере шаблон поиска выглядит так: *пробельный символ один раз*.

'1 12 123 abc @@@'.replace(/\S+/g, '!'); //вернет '! ! ! ! !'

В данном примере шаблон поиска выглядит так: *НЕ пробельный символ один или более раз*. Все подстроки, разделенные пробелами, заменятся на '!'.

'1 12 123a Abc @@@'.replace(/\w+/g, '!'); //вернет '! ! ! ! @@@'

В данном примере шаблон поиска выглядит так: *цифра или буква один или более раз*. Все подстроки, состоящие из цифр и букв, заменятся на '!'.

'1 12 123 Abc @@@'.replace(/\W+/g, '!'); //вернет '1!12!123!Abc!'

В данном примере шаблон поиска выглядит так: *НЕ цифра и НЕ буква один или более раз*. Под данное определение в нашем случае попадает '@@@' и все пробелы (они ведь тоже не цифры и не буквы!). Обратите внимание на то, что в конце один '!' - в него преобразовалась строка ' @@@' – с пробелом впереди.

## Квадратные скобки '[' и ']'

Группы символов **\s, \S, \w, \W, \d, \D** не очень гибкие. Даже такая простая задача, как *найти все буквы (но не цифры)* не может быть решена ими. Давайте посмотрим, что еще могут предложить нам регулярные выражения.

А предлагают они нам квадратные скобки **'['** и **']'**, которые представляют собой операцию **'или'**:

'axx bxx cxx exx'.replace(/[abc]xx/g, '!'); //вернет '! ! ! exx'

В данном примере шаблон поиска выглядит так: *первый символ - это буква 'a', 'b' или 'c', потом две буквы 'x'*.

Но этим их возможности далеко не исчерпаны. С помощью шляпки **'^'** мы можем сделать отрицание:

'axx bxx cxx exx'.replace(/[^abc]xx/g, '!'); //вернет 'axx bxx cxx !'

В данном примере шаблон поиска выглядит так: *первый символ - это НЕ буква 'a', 'b' или 'c' (любой символ кроме них), потом две буквы 'x'*.

### Что можно еще:

* Можно задавать группы символов: **[a-z]** задаст маленькие латинские буквы, **[A-Z]** – большие, **[0-9]** – цифру от 0 до 9.
* Посложнее: **[a-zA-Z]** – большие и маленькие латинские буквы, то же самое плюс цифры - **[a-zA-Z0-9]**, и так далее. Порядок значения не имеет, нет разницы **[a-zA-Z]** или **[A-Za-z]**.
* Еще посложнее: **[2-5]** - цифра от 2-х до 5-ти (обе включительно), **[a-c]** – буквы от 'a' до 'c' по алфавиту (то есть 'a', 'b', 'c') и так далее.

### Особенности:

. Шляпка - это спецсимвол внутри **[ ]** (снаружи, кстати, тоже). Если вам нужна **шляпка как символ** - просто поставьте ее не в начале (она спецсимвол только вначале: **[^d]** – так спецсимвол, а **[d^]** - так уже нет!).

'axx bxx ^xx dxx'.replace(/[^d]xx/g, '!'); //вернет '! ! ! dxx'

Шаблон поиска такой: *первый символ - это все кроме 'd', потом две буквы 'x'*.

'axx bxx ^xx dxx'.replace(/[d^]xx/g, '!'); //вернет 'axx bxx ! !'

А теперь так: *первый символ - это 'd' или '^', потом две буквы 'x'*.

Можно не убирать шляпку с первого места, а просто заэкранировать ее с помощью обратного слеша, и она станет обозначать саму себя:

'axx bxx ^xx dxx'.replace(/[\^d]xx/g, '!'); //вернет 'axx bxx ! !'

Шаблон поиска такой: *первый символ - это 'd' или '^', потом две буквы 'x'*.

. Дефис - тоже спецсимвол внутри **[ ]** (а вот снаружи - нет!). Если вам нужен сам дефис как символ - то поставьте его там, где он **не будет воспринят как разделитель группы**.

**Почему это важно:** вы можете сделать группу символов, сами не заметив этого. К примеру, вот так - **'[:-@]'** - вы думаете, что выбираете двоеточие, дефис и собаку @, а на самом деле получается группа символов между : и @. В эту группу входят следующие символы: **":, ;, <, =, >, ?"**.

Откуда они взялись? Из таблицы ASCII - двоеточие имеет номер меньше, чем собака - и получается группа. То есть все группы получаются по таблице ASCII (при желании этим можно пользоваться).

Как с этим бороться: поставьте символ дефиса там, где он точно не будет воспринят как символ группы, например, в начале или в конце (то есть после **[** или перед **]**):

'axx Axx -xx @xx'.replace(/[a-zA-Z-]xx/g, '!'); //вернет '! ! ! @xx'

Шаблон поиска такой: *первый символ - это маленькие, большие буквы или дефис '-', потом две буквы 'x'*.

Можно сделать и так:

'axx 9xx -xx @xx'.replace(/[a-z-0-9]xx/g, '!'); //вернет '! ! ! @xx'

Шаблон поиска такой: *первый символ - это маленькие латинские, дефис '-' или цифра от 0 до 9, потом две буквы 'x'*.

Это работает, так как символы **'z-0'** не могут образовать группу, ведь буква **'z'** уже входит в группу **'a-z'**. Но лучше так не рисковать (мало ли что), а просто ставить дефис в начале или в конце.

Можно также заэкранировать дефис - тогда он будет обозначать сам себя независимо от позиции. Например, вместо **'[:-@]'** написать **'[:\-@]'** - и группы уже не будет, а будут три символа - двоеточие, дефис и собака @.

. Спецсимволы внутри **[ ]** становятся обычными символами (значит их не надо экранировать обратным слешем!).

. Исключение из предыдущего: **[ ]** имеют свои спецсимволы - это **'^'** и **'-'**, кроме того, если вам нужны квадратные скобки как символы внутри **[ ]** - то их нужно экранировать обратным слешем.

. Еще исключение: группы символов **\s, \S, \w, \W, \d, \D** (и другие аналогичные) будут обозначать именно группы, то есть по-прежнему будут командами.

'3xx axx Axx'.replace(/[\da-z]xx/g, '!'); //вернет '! ! Axx'

Шаблон поиска такой: *первый символ - это цифра от 0 до 9 (\d) или маленькая латинская буква (a-z), потом две буквы 'x'*.

. Операторы повторений записываются сразу после **[ ]** (их не надо брать в круглые скобки):

'.xx ..xx .a.xx bxx'.replace(/[.a]+xx/g, '!'); //вернет '! ! ! bxx'

Шаблон поиска такой: *точка или буква 'a' один или более раз, потом две буквы 'x'*.

Иногда один и тот же эффект можно получить и группами \s, \S, \w, \W, \d, \D, и скобками []. Когда пользоваться группами, а когда []?

Пользуйтесь тем, что записывается короче и выглядит проще: **\w** проще, чем **[a-zA-Z]**, а **\d** – короче, чем **[0-9]**. Хотя вторые варианты могут быть нагляднее (**[0-9]** сразу понятно, а **\d** – еще нужно вспомнить, что это).

Но в случае 'цифры от 2-x до 5-ти' - можно сделать только так: **[2-5]** или, что длиннее, можно и так: **[2345]**. Тоже вариант, но уж очень длинно.

## Особенности кириллицы

**Первое:** кириллица не входит в **\w**, нужно делать так: **[а-яА-Я]**.

**Второе:** так сделать недостаточно **[а-яА-Я]** - сюда не войдет буква ё, нужно делать так: **[а-яА-ЯЁё]**.

Если нужны только маленькие буквы - тогда просто **[а-яё]**.

'аяя ёяя 2яя'.replace(**/**[а**-**яА**-**ЯЁё]яя#', '**!**'); //вернет '! ! 2яя'

Шаблон поиска такой: *первый символ - это цифра кириллическая буква, потом две буквы 'я'*.

## Начало '^' и конец '$' строки

Существуют специальные символы, которые обозначают начало **'^'** или конец строки **'$'**. Посмотрите примеры:

'aaa aaa aaa'.replace(/^aaa/g, '!'); //вернет '! aaa aaa'

Шаблон поиска такой: *заменить 'aaa' на '!' только если оно стоит в начале строки*.

'aaa aaa aaa'.replace(/aaa$/g, '!'); //вернет 'aaa aaa !'

Шаблон поиска такой: *заменить 'aaa' на '!' только если оно стоит в конце строки*.

'aaa'.replace(/^a+$/g, '!'); //вернет '!'

Шаблон поиска такой: *буква 'a' повторяется один или более раз, заменить всю строку на '!' только она состоит из одних букв 'a'*.

Обратите внимание на этот прием: когда в начале регулярки стоит '^', а в конце - '$'. Таким образом мы проверяем **всю** строку целиком на соответствие регулярке.

## 'Или' через вертикальную черту |

Квадратные скобки не единственный вариант сделать **'или'**: существует еще вариант через вертикальную черту **'|'**:

'bbbb'.replace(/a|b+|c/g, '!'); //вернет '!'

Шаблон поиска такой: *если вся строка - это 'a', или вся строка - один или более букв 'b', или вся строка - это 'c', то заменить ее на '!'*.

В данном случае 'или' действует на все регулярное выражение (по сути у нас три регулярки в одной). Можно работать и по-другому - поставим **круглые скобки**, и теперь '|' будет действовать только внутри них:

'axx bxx bbxx exx'.replace(/(a|b+)xx/g, '!'); //вернет '! ! ! exx'

Шаблон поиска такой: *в начале стоит или 'a', или 'b' один или более раз, а потом две буквы 'x'*.

Данный вариант 'или' отличается от квадратных скобок - он мощнее. Такие вещи, как в последнем примере (в начале 'a' или 'b' один или более раз) через **[ ]** не сделать.

Для простых вещей **'|'** лучше не использовать: сравните **(a|b|c)** или **[abc]** – второй вариант короче.

## Проблема обратного слеша

Обратный слеш сам по себе является спецсимволом JavaScript. Он нужен в таких случаях: **var str = 'д'Артаньян'** - я хочу поставить одиночную кавычку внутри строки, которая сама в одиночных кавычках. У меня ничего не выйдет - JavaScript выдаст ошибку.

Чтобы поставить *кавычку как символ* в нашем случае - я должен экранировать ее обратным слешем: **var str = 'д\'Артаньян'** – вот так будет работать.

А если я хочу поставить *обратный слеш как символ* вовнутрь строки JavaScript? Придется экранировать его самого: **var str = 'обратный слеш \\ - это он'**. В результате здесь он будет обозначать сам себя.

Теперь перейдем к регуляркам: обратный слеш является спецсимволом JavaScript и спецсимволом регулярок. Чаще всего нам не нужно удваивать этот слеш для JavaScript (например, мы пишем \s, а не \\s), однако, если мы хотим поставить *обратный слеш как символ* - нам придется написать его аж **4 раза** (2 раза для JavaScript и два раза для регулярки)!

'\\ \\ \\\\'.replace(/\\\\/g, '!'); //вернет '! ! !!'

Шаблон поиска такой: *обратный слеш один раз*.

Обратите внимание на **'\\ \\ \\\\'** - мы удваиваем все слеши для JavaScript, и в реальности строка выглядит так: '\ \ \\', поэтому в ответе будет '! ! !!', а не '!! !! !!!!'.

Обратите внимание: здесь **+** действует аж на 4 символа перед ним без группировки:

'\\ \\ \\\\'.replace(/\\\\+/g, '!'); //вернет '! ! !'

Шаблон поиска такой: *обратный слеш один или более раз*.

**Метод test**

Рассмотрим метод **test**, который проверяет, есть ли в строке хотя бы одно совпадение с регуляркой. Если есть - возвращается true, а если нет - false.

Метод работает так: **регулярка.test(где искать)** - то есть порядок параметров не такой, как в replace.

Смотрите пример:

/a+/.test('eee aaa bbb'); //вернет true

Шаблон поиска такой: *буква 'a' один или более раз*.

preg\_match(/a+/g, 'eee bbb'); //вернет 0

Метод ничего не нашел - вернет **false**.

Часто данный метод используется для проверки на соответствие регулярному выражению целой строки. Например, мы хотим узнать - данная строка **корректный email** или нет:

/^[a-zA-Z-.]+@[a-z]+\.[a-z]{2,3}$/.test('my-mail@mail.ru');

Шаблон поиска такой:

**[a-zA-Z-.]+** маленькие или большие латинские буквы, точка или '-' 1 или более раз, @ потом @,

**[a-z]+** потом маленькие латинские 1 или более раз,

**\.** потом точка,

**[a-z]{2,3}** потом маленькие латинские два или три раза (ru, by, com и т.п.).

В результате мы получим **true** - наша строка будет корректным емэйлом.

А вот так получим false:

/^[a-zA-Z-.]+@[a-z]+\.[a-z]{2,3}$/.test('my-#mail@mail.ru');

Обратите внимание на то, что мы вначале регулярки ставим **^**, а в конце **$** - этим мы говорим, что под шаблон должна **попасть вся строка**. Если их не поставить - тогда мы скажем не **'вся строка есть email'**, а **'в строке есть email'**:

/[a-zA-Z-.]+@[a-z]+\.[a-z]{2,3}/.test('#$%my-mail@mail.ru&@$');

Метод вернет **false** - он нашел одно совпадение (выделено голубым). Но переданная строка отнюдь не корректный email (она просто содержит его внутри, но по краям - мусор).

**Метод match с модификатором g**

Давайте теперь разберем следующий полезный метод - метод **match**. Он позволяет получить ту часть строки, которая попала под регулярное выражение.

Этот метод работает по-разному в зависимости от того, есть модификатор g или нет. Если он есть - метод возвращает массив подстрок, которые попали под регулярное выражение.

Метод работает так: **где\_искать.match(регулярка)**.

Давайте рассмотрим его работу на следующем примере:

'a aa aaa aaaa'.match(/a+/g); //вернет ['a', 'aa', 'aaa', 'aaaa']

Шаблон поиска такой: *буква 'a' один или более раз*.

В случае, если совпадений не было, метод match возвращает **null**, а не пустой массив. Обратите на это внимание.

**Метод match без модификатора g**

Если вызвать метод **match** без модификатора g, то он найдет только первое совпадение с регуляркой.

Однако, вернет он все равно массив - это будет массив, состоящий из найденного совпадения, с дополнительными свойствами: **index** – позиция, на которой оно обнаружено и **input** – строка, в которой был поиск (в принципе последнее - бесполезная штука).

А **найденная подстрока** будет лежать в нулевом элементе возвращенного массива (почему так - поймете дальше).

Как это работает - изучите на следующим примере:

var result **=** 'aaa bbb ccc'.match(/b+/);

alert(result[0]); //выведет 'bbb' - то, что попало под регулярку

alert(result.index); //выведет 4 - позиция начала 'bbb' в строке

alert(result.input); //выведет 'aaa bbb ccc' - исходную строку

Шаблон поиска такой: *буква 'b' один или более раз*.

**Карманы для метода match без модификатора g**

Мы с вами использовали круглые скобки **( )** для группировки, однако они имеют еще одно очень важно применение - используются в качестве *карманов*.

Чтобы разобраться с этим непростым понятием - смотрите пример:

var result **=** 'bbb xaaax ccc'.match(/x(a+)x/);

alert(result[0]); //выведет 'xaaax' - то, что попало под регулярку

alert(result[1]); //выведет 'aaa' - то, что попало в карман

Шаблон поиска такой: *буква 'x', затем 'a' один или более раз, затем буква 'x'*.

В **result[0]** попало то, что нашла регулярка, а в **result[1]** - то, что нашла регулярка в круглых скобках.

То есть **карман** - это такой способ хранения части того, что мы ищем. Например, мы ищем домены вида **domain.ru**, но хотим узнать только доменную зону (ru, com и т.п.).

В регулярке придется указать, что нам нужны строки вида **'domain.ru'** (иначе мы ничего не найдем - мы не можем просто искать доменные зоны, так как непонятно как их найти, нужно привязаться к доменам), но раз нас интересует только зона - то **положим ее в карман**. Давайте посмотрим на примере:

var result **=** 'domain.ru'.match(/[a-z]+\.([a-z]{2,3})/);

Шаблон поиска такой: **[a-z]+** - маленькие буквы один или более раз (domain, site и т.п.) **\.** - точка **([a-z]{2,3})** - маленькие буквы 2 или 3 раза (ru, com, by, net и т.п.)

Посмотрим содержимое result: в **result[0]** попало то, что нашла регулярка, то есть 'domain.ru', а в **result[1]** - содержимое кармана, то есть 'ru'.

Можно использовать **не один карман**, а несколько. Положим в первый карман имя домена, а во второй - зону:

var result **=** 'domain.ru'.match(/([a-z]+)\.([a-z]{2,3})/);

alert(result[0]); //выведет 'domain.ru' - то, что попало под регулярку

alert(result[1]); //выведет 'domain' - то, что попало в первый карман

alert(result[2]); //выведет 'ru' - то, что попало во второй карман

Карманы нумеруются по порядку в регулярке: **/([a-z]+)\.([a-z]{2,3})/** - первые круглые скобки - первый карман, вторые - второй карман и так далее.

**Карманы внутри replace**

Карманы можно использовать и при работе с методом **replace** - то, что мы положим в карман, затем может быть использовано во втором параметре: **где\_заменить.replace(регулярка с карманом, на что заменить)**.

Если мы что-то положим в карман в регулярке, то в параметре **'на что заменить'** мы можем обратиться к этому карману так: **$1** – первый карман, **$2** второй карман и так далее.

Давайте решим следующую задачу: даны строки вида **'aaa@bbb'** - буквы, потом собака, потом буквы. Нужно поменять местами буквы до @ и после. В нашем случае из **'aaa@bbb'** сделать **'bbb@aaa'**:

'a@b aa@bb'.replace(/([a-z]+)@([a-z]+)/g, '$2@$1'); //b@a bb@aa

Шаблон поиска такой: *маленькие буквы один или более раз (первый карман), собака, и опять маленькие буквы один или более раз (второй карман)*.

Шаблон замены такой: *мы говорим: замени найденное на* ***$2@$1****, где* ***$2*** *– содержимое второго кармана, а* ***$1*** *- первого*.

Давайте разберем подстроку **'a@b'**, что с ней происходит: **'a'** ложится в первый карман и доступно как **$1**, **'b'** ложится во второй карман и доступно как **$2**.

При замене **$2@$1** мы говорим: **$2** (вставится **'b'**), собака **@**, **$1** (вставится **'a'**).

Карман **$0** соответствует **всему выражению**. Давайте заменим подстроки из букв на них самих с '!' по краям:

'aaa bbb'.replace(/[a-z]+/g, '!$0!'); //вернет '!aaa! !bbb!'

**!$0!** - **$0** это найденная строка (сначала **'aaa'**, потом **'bbb'**). Мы говорим: *замени 'aaa' на ее саму ($0), но с '!' по краям !$0!.* И получаем '!aaa!'. Для **'bbb'** аналогично.

**Карманы по умолчанию внутри replace**

Вы уже знаете, что по умолчанию всегда есть карман $0, в котором лежит попавшая под регулярку подстрока. Однако, в строке замены по умолчанию также доступно еще несколько команд:

* **$`** - часть строки до совпадения.
* **$'** - часть строки после совпадения.
* **$&** - всё найденное совпадение (равно $0).
* **$$** - знак доллара.

**Несохраняющие скобки**

К сожалению, скобки **( )** выполняют две функции - группировка символов и функцию кармана. А что делать, если нам нужно сгруппировать, но в карман не ложить? Для этого придуманы специальные **несохраняющие скобки** (выделены красным) (?:a+) - они группируют, но не ложат в карман:

'ababx abe'.replace(/(?:ab)+([a-z])/g, '!$1!'); //вернет '!x! !e!'

Шаблон поиска следующий: *'ab' один или более раз, затем одна буква, которую ложим в карман*.

Шаблон замены: *заменим найденное на первый карман, обернутый '!' справа и слева*.

Так как первый карман - это **([a-z])**, то в него попадет сначала **'x'**, а потом **'e'**.

А вот если мы вместо несохраняющих скобок возьмем **обычные**, то первый карман будет **(ab)**:

'ababx abe'.replace(/(ab)+([a-z])/g, '!$1!'); //вернет '!ab! !ab!'

Обратите внимание: **+** не размножает карманы - **'abab'** превратится в '!', а не '!!'.

Но мы-то хотели, чтобы в карман попадало не **'ab'**, а то, что после него. Это будет уже второй карман:

'ababx abe'.replace(/(ab)+([a-z])/g, '!$2!'); //вернет '!x! !e!'

Получается немного неудобно - **$1** (первый карман) **нигде не используется**. А начинаем сразу с **$2**. Чтобы не было таких неудобств - и придуманы скобки **(?: )**.

Хотя, если вас это не смущает, - можно их и не использовать.

**Исключение:** когда у вас много группировок, а карманов мало - будет неудобно считать все скобки, чтобы понять, что вам нужно $5 и $9, а остальные карманы просто группировка. А если что-то поменяется - то придется все пересчитывать. Тут точно лучше использовать **(?: )** для группировки, а **( )** - только для карманов.

## Позитивный и негативный просмотр

Иногда нужно решить задачу такого типа: найти строку 'aaa' и заменить ее на '!', **но только если после 'aaa' стоит 'x'** (при этом 'x' не заменять).

Если мы попытаемся решить задачу '**в лоб**', то у нас ничего не выйдет:

'aaax baaa'.replace(/xaaa/g, '!'); //вернет '! baaa', а хотели '!x baaa'

Нужен способ сказать, что *'x' не следует заменять*. Делается это с помощью специальных скобок **(?= )**, которые просто смотрят, но **не забирают с собой**:

'xaaa baaa'.replace(/aaa(?=x)/g, '!'); //вернет 'x! baaa'

Скобки **(?= )** называются **позитивный просмотр вперед**. Позитивный - так как 'x' (в нашем случае) должен быть - только тогда произойдет замена. Есть и **негативный просмотр вперед** - **(?! )** - он, наоборот, говорит, что чего-то должно не быть:

//Если перед 'aaa' стоит НЕ 'x', тогда заменим на '!':

'xaaa baaa'.replace(/aaa(?!x)/g, '!'); //вернет 'xaaa b!'

К сожалению, в JavaScript, в отличие от других языков программирования, просмотра назад нет.

## Метод replace - продвинутое использование

Метод replace вторым параметром может принимать не только строку, но и функцию, которая применится для каждого найденного совпадения.

Каждая подстрока, которую нашла регулярка, заменится на то, что вернет эта функция именно для этой подстроки.

В эту функцию можно передавать параметры: в первый параметр положится найденная строка, во второй параметр - первый карман, в третий параметр - второй карман и так далее - можно делать сколько параметров, сколько карманов в регулярном выражении.

В предпоследний параметр положится позиция найденного совпадения, а в последний - вся строка, по которой делается поиск.

где\_ищем.replace(регулярка, function (найденное, карман1, карман2..., позиция, где ищем) {

});

Как это все работает - разберем на практических примерах. Давайте решим следующую задачу:

**Задача:** в строке даны числа, нужно заменить их на квадраты этих чисел. Для начала поэкспериментируем:

'2 3 4 5'.replace(/\d+/g, function (match) {

alert(match);

});

Функция **alert** последовательно выведет 2, потом 3, 4 и 5. То есть в переменную match последовательно ложатся те строки, которые нашла регулярка.

Давайте решим задачу до конца - будем возводить **match** в квадрат и возвращать его с помощью return. Получится, что для найденной двойки вернется 4 и двойка заменится на эту четверку, для найденной тройки вернется 9 и тройка заменится на эту девятку - и так далее:

var result **=** '2 3 4 5'.replace(/\d+/g, function (match) {

return match**\***match;

});

alert(result);

Функция **alert** выведет '4 9 16 25' - то, что мы и хотели.

**Еще задача:** в строке даны конструкции вида '2+3=', нужно после равно вставить сумму этих чисел. Сама строка выглядит примерно так: '2+3= ... 25+50='. Опять давайте поэкспериментируем - разложим первое и второе слагаемые по отдельным карманам:

'2+3= ... 25+50='.replace(/(\d+)\+(\d+)=/g, function (match, match1, match2) {

alert(match);

alert(match1);

alert(match2);

});

Сейчас мы увидим 6 алертов: первая тройка - это '2+3=', потом '2' (первый карман), потом '3' (второй карман), затем вторая тройка (для второго совпадения) - это '25+50=', потом '25' (первый карман), потом '50' (второй карман).

Давайте теперь окончательно решим задачу: для каждой найденной подстроки просуммируем первый и второй карманы, возьмем нулевой карман (найденную строку, например '2+3='), добавим к нему результат и вернем все это через return:

var result **=** '2+3= ... 25+50='.replace(/(\d+)\+(\d+)=/g, function (match, match1, match2) {

var sum **=** match1 **+** match2;

return match**+**sum; //здесь к '2+3=' прибавляется 5 - складываем как строки, для 25+50 аналогично

});

alert(result);

В результате alert выведет такую строку '2+3=5 ... 25+50=75' - для каждой найденной конструкции будет вычислена сумма.

## Еще методы для работы с регулярками

*Ниже мы разберем еще 3 полезных метода работы с регулярками: search, split и exec.*

### Метод search

Метод **search** принимает следующие параметры: *где\_ищем.replace(регулярка)*, а возвращает позицию первой найденной подстроки (если она не найдена - то вернет -1).

### Метод split

Метод split уже хорошо вам знаком - он разбивает строку в массив по разделителю. Так вот, параметром он может принимать не только разделитель, но и регулярное выражение. В этом случае разделителем будет выступать все подстроки, которые попали под регулярное выражение.