Регулярные выражения

regexp = new RegExp("шаблон", "флаги");

regexp = /шаблон/; // без флагов

regexp = /шаблон/gmi; // где флаги:

- i - поиск не зависит от регистра
- g - поиск ищет все совпадения, без него – только первое.
- m - многострочный режим.
- s - включает режим «dotall», при котором точка . может соответствовать символу перевода строки \n
- u- включает полную поддержку юникода. Флаг разрешает корректную обработку суррогатных пар
- y - режим поиска на конкретной позиции в тексте

Слеши /.../ не допускают никаких вставок переменных. Используются, когда мы на момент написания кода точно знаем, каким будет регулярное выражение – и это большинство ситуаций. А new RegExp – когда мы хотим создать регулярное выражение «на лету» из динамически сгенерированной строки.

str.match

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

У него есть три режима работы:

- Если у регулярного выражения есть флаг g, то он возвращает массив всех совпадений.
- Если такого флага нет, то возвращает только первое совпадение в виде массива, в котором по индексу 0 находится совпадение, и есть свойства с дополнительной информацией о нём:

```
let str = "Любо, братцы, любо!";
let result = str.match(/любо/i); // без флага g
alert( result[0] );     // Любо (первое совпадение)
alert( result.length ); // 1
alert( result.index );  // 0 (позиция совпадения)
alert( result.input );  // Любо, братцы, любо! (исходная строка)
```

- И, наконец, если совпадений нет, то, вне зависимости от наличия флага g, возвращается null. Если хочется, чтобы результатом всегда был массив, можно написать так:

```
let matches = "JavaScript".match(/HTML/) || [];
```

str.replace(str|regexp, str|func)

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

```
alert( "We will, we will".replace(/we/i, "I") ); // I will, we will
alert( "We will, we will".replace(/we/ig, "I") ); // I will, I will
```

В строке замены replacement мы можем использовать специальные комбинации символов для вставки фрагментов совпадения:

- $& вставляет всё найденное совпадение
- $` вставляет часть строки до совпадения
- $' вставляет часть строки после совпадения
- $n если n это 1-2 значное число, вставляет содержимое n-й скобочной группы регулярного выражения,
- $<name> вставляет содержимое скобочной группы с именем name
- $$ вставляет символ "$"

```
alert( "Люблю HTML".replace(/HTML/, "$& и JavaScript") ); // Люблю HTML и JavaScript
```

Для ситуаций, которые требуют «умных» замен, вторым аргументом может быть **функция.**

Она будет вызываться для каждого совпадения, и её результат будет вставлен в качестве замены.

Функция вызывается с аргументами **func(match, p1, p2, ..., pn, offset, input, groups)**

match – найденное совпадение,

p1, p2, ..., pn – содержимое скобок (см. главу Скобочные группы).

offset – позиция, на которой найдено совпадение,

input – исходная строка,

groups – объект с содержимым именованных скобок (см. главу Скобочные группы).
Если скобок в регулярном выражении нет, то будет только 3 аргумента: func(match, offset, input).
Например, переведём выбранные совпадения в верхний регистр:

```
let str = "html and css";
let result = str.replace(/html|css/gi, str => str.toUpperCase());
alert(result); // HTML and CSS
Заменим каждое совпадение на его позицию в строке:
alert("Xo-Xo-xo".replace(/xo/gi, (match, offset) => offset)); // 0-3-6
```

regexp.exec(str)

Метод regexp.exec(str) ищет совпадение с regexp в строке str. В отличие от предыдущих методов, вызывается на регулярном выражении, а не на строке. Он ведёт себя по-разному в зависимости от того, имеет ли регулярное выражение флаг g.
Если нет g, то regexp.exec(str) возвращает первое совпадение в точности как str.match(regexp). Такое поведение не даёт нам ничего нового.
Но если есть g, то:
Вызов regexp.exec(str) возвращает первое совпадение и запоминает позицию после него в свойстве regexp.lastIndex.
Следующий такой вызов начинает поиск с позиции **regexp.lastIndex**, возвращает следующее совпадение и запоминает позицию после него в regexp.lastIndex.
…И так далее.
Если совпадений больше нет, то regexp.exec возвращает null, а для regexp.lastIndex устанавливается значение 0.
Таким образом, повторные вызовы возвращают одно за другим все совпадения, используя свойство regexp.lastIndex для отслеживания текущей позиции поиска.

Мы можем использовать regexp.exec для поиска совпадения, начиная с нужной позиции, **если вручную поставим lastIndex**.

```
Например:
let str = 'Hello, world!';
let regexp = /\w+/g; // без флага g свойство lastIndex игнорируется
regexp.lastIndex = 5; // ищем с 5-й позиции (т.е с запятой и далее)
alert( regexp.exec(str) ); // world
```

regexp.test

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

```
let str = "Я ЛюБлЮ JavaScript";
let regexp = /люблю/i;
alert( regexp.test(str) ); // true
```

Если регулярное выражение имеет флаг g, то regexp.test ищет, начиная с **regexp.lastIndex** и обновляет это свойство, аналогично regexp.exec. Таким образом, мы можем использовать его для поиска с заданной позиции:

```
let regexp = /люблю/gi;
let str = "Я люблю JavaScript";
// начать поиск с 10-й позиции:
regexp.lastIndex = 10;
alert( regexp.test(str) ); // false (совпадений нет)
```

matchAll

Он был добавлен в язык JavaScript гораздо позже чем str.match, как его «новая и улучшенная» версия.
Он, как и str.match(regexp), ищет совпадения, но у него есть три отличия:

- Он возвращает не массив, а перебираемый объект.
- При поиске с флагом g, он возвращает каждое совпадение в виде массива со скобочными группами.
- Если совпадений нет, он возвращает не null, а просто пустой перебираемый объект.

```
Например:
let str = '<h1>Hello, world!</h1>';
let regexp = /<(.*?)>/g;
let matchAll = str.matchAll(regexp);
```

```
alert(matchAll); // [object RegExp String Iterator], не массив, а перебираемый объект
matchAll = Array.from(matchAll); // теперь массив
let firstMatch = matchAll[0];
alert( firstMatch[0] );  // <h1>
alert( firstMatch[1] );  // h1
alert( firstMatch.index );  // 0
alert( firstMatch.input );  // <h1>Hello, world!</h1>
```

str.split(regexp|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('-')) // массив [12, 34, 56]
```

Но мы можем разделить по регулярному выражению аналогичным образом:

```
alert('12, 34, 56'.split(/,\s*/)) // массив [12, 34, 56]
```

str.search(regexp)

Метод str.search(regexp) возвращает позицию первого совпадения с regexp в строке str или -1, если совпадения нет.

Например:

```
let str = "Я люблю JavaScript!";
let regexp = /Java.+/;
alert( str.search(regexp) ); // 8
```

Символьные классы, Обратные символьные классы

\d - цифра: символ от 0 до 9.
\s - пробельные символы: символ пробела, табуляции \t, перевода строки \n и некоторые другие редкие пробельные символы, обозначаемые как \v, \f и \r.
\w - символ «слова», а точнее – буква латинского алфавита или цифра или подчёркивание _. Нелатинские буквы не являются частью класса \w, то есть буква русского алфавита не подходит.
\D - не цифра: любой символ, кроме \d, например буква.
\S - не пробел: любой символ, кроме \s, например буква.
\W - любой символ, кроме \w, то есть не буквы из латиницы, не знак подчёркивания и не цифра. В частности, русские буквы принадлежат этому классу.

Точка . – это специальный символьный класс, который соответствует любому символу, кроме новой строки (но не «отсутствие символа). Если регулярное выражение имеет s, то точка . соответствует буквально любому символу.

Якоря: каретка **^** означает совпадение с началом текста, а доллар **$** – с концом. Оба якоря вместе ^...$ часто используются для проверки, совпадает ли строка с шаблоном полностью. Якоря ^ и $ – это проверки. У них нулевая ширина. Другими словами, они не добавляют к результату поиска символы, а только заставляют движок регулярных выражений проверять условие (начало/конец текста).

Многострочный режим включается флагом **m**. Он влияет только на поведение ^ и $. В многострочном режиме они означают не только начало/конец текста, но и начало/конец каждой строки в тексте.
Для того, чтобы найти конец строки, можно использовать не только якоря ^ и $, но и символ перевода строки **\n**. В отличие от якорей ^ $, которые только проверяют условие (начало/конец строки), \n – символ и входит в результат.

Граница слова \b – проверка, как ^ и $. Есть три вида позиций, которые являются границами слова:

- Начало текста, если его первый символ \w.
- Позиция внутри текста, если слева находится \w, а справа – не \w, или наоборот.
- Конец текста, если его последний символ \w.

\w означает латинскую букву (или цифру или знак подчёркивания), поэтому проверка не будет работать для других символов, например, кириллицы или иероглифов).

Символы, которые нужно экранировать: [\ ^ $. | ? * + ( ). При передаче строки в **new RegExp** нужно удваивать обратную косую черту: **** для экранирования специальных символов, потому что строковые кавычки «съедят» одну черту. , (например: \n – становится символом перевода строки,
\u1234 – становится символом Юникода с указанным номером, а когда нет особого значения: как например для \d или\z, обратная косая черта просто удаляется.)

Наборы и диапазоны [...]

Несколько символов или символьных классов в квадратных скобках **[…]** означают «искать любой символ из заданных». Ещё квадратные скобки могут содержать **диапазоны** символов. (\d – то же самое, что и [0-9],
\w – то же самое, что и [a-zA-Z0-9_], \s – то же самое, что и [\t\n\v\f\r], плюс несколько редких пробельных символов Юникода.). Помимо обычных диапазонов, есть **«исключающие» диапазоны**, которые выглядят как **[^…]**.
В квадратных скобках большинство специальных символов можно использовать без экранирования:

- Символы . + ( ) не нужно экранировать никогда.
- Тире - не надо экранировать в начале или в конце (где оно не задаёт диапазон).
- Символ каретки ^ нужно экранировать только в начале (где он означает исключение).
- Закрывающую квадратную скобку], если нужен именно такой символ, экранировать нужно

Квантификаторы

{n} - он добавляется к символу (или символьному классу, или набору [...] и т.д.) и указывает, сколько их нам нужно. Шаблон **\d{5}** обозначает ровно 5 цифр, он эквивалентен \d\d\d\d\d. Для того, чтобы найти числа от 3 до 5 цифр, мы можем указать границы в фигурных скобках: **\d{3,5}**

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

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

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

Скобочные группы

Часть шаблона можно заключить в скобки (...). У такого выделения есть два эффекта:

- Позволяет поместить часть совпадения в отдельный массив.
- Если установить квантификатор после скобок, то он будет применяться ко всему содержимому скобки, а не к одному символу.

Метод **str.match(regexp),** если у регулярного выражения regexp нет флага g, ищет первое совпадение и возвращает его в виде массива:

- На позиции 0 будет всё совпадение целиком.
- На позиции 1 – содержимое первой скобочной группы.
- На позиции 2 – содержимое второй скобочной группы. …и так далее…

```
let str = '<h1>Hello, world!</h1>';
let tag = str.match(/<(.*?)>/);
alert( tag[0] ); // <h1>
alert( tag[1] ); // h1
```

Скобки могут быть и вложенными. Даже если скобочная группа необязательна (например, стоит квантификатор (...)?), соответствующий элемент массива result существует и равен undefined.

Именованные группы

Запоминать группы по номерам не очень удобно. Для простых шаблонов это допустимо, но в сложных регулярных выражениях считать скобки затруднительно. Гораздо лучше – давать скобкам имена. Это делается добавлением **?<name>** непосредственно после открытия скобки.

Например, поищем дату в формате «день-месяц-год»:

```
let dateRegexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/;
let str = "2019-04-30";
let groups = str.match(dateRegexp).groups;
alert(groups.year); // 2019
alert(groups.month); // 04
alert(groups.day); // 30
```

Скобочные группы при замене

Метод str.replace(regexp, replacement), осуществляющий замену совпадений с regexp в строке str, позволяет использовать в строке замены содержимое скобок. Это делается при помощи обозначений вида $n, где n – номер скобочной группы.

Например:

```
let str = "John Bull";
let regexp = /(\w+) (\w+)/;
alert( str.replace(regexp, '$2, $1') ); // Bull, John
```

Для именованных скобок ссылка будет выглядеть как $<имя>.

Например, заменим даты в формате «год-месяц-день» на «день.месяц.год»:

```
let regexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/g;
let str = "2019-10-30, 2020-01-01";
alert( str.replace(regexp, '$<day>.$<month>.$<year>') );
// 30.10.2019, 01.01.2020
```

Исключение из запоминания через ?:

Бывает так, что скобки нужны, чтобы квантификатор правильно применился, но мы не хотим, чтобы их содержимое было выделено в результате. Скобочную группу можно исключить из запоминаемых и нумеруемых, добавив в её начало **?:**

Альтернация (или) |

Альтернация – термин в регулярных выражениях, которому в русском языке соответствует слово «ИЛИ».

В регулярных выражениях она обозначается символом вертикальной черты **|**

Например, нам нужно найти языки программирования: HTML, PHP, Java и JavaScript.

Соответствующее регулярное выражение: html|php|java(script)?.

```
let regexp = /html|css|java(script)?/gi;
let str = "Сначала появился язык Java, затем HTML, потом JavaScript";
alert( str.match(regexp) ); // Java,HTML,JavaScript
```

Опережающие и ретроспективные проверки

В некоторых случаях нам нужно найти соответствия шаблону, но только те, за которыми или перед которыми следует другой шаблон. Для этого в регулярных выражениях есть специальный синтаксис: опережающая (lookahead) и ретроспективная (lookbehind) проверка.

В качестве первого примера найдём стоимость из строки 1 индейка стоит 30€. То есть, найдём число, после которого есть знак валюты €.

Опережающая проверка

Синтаксис опережающей проверки: **X(?=Y)**

Он означает: найди X при условии, что за ним следует Y. Вместо X и Y здесь может быть любой шаблон.

Для целого числа, за которым идёт знак €, шаблон регулярного выражения будет \d+(?=€):

```
let str = "1 индейка стоит 30€";
alert( str.match(/\d+(?=€)/) ); // 30, число 1 проигнорировано, так как за ним НЕ следует €
```

Возможны и более сложные проверки, например **X(?=Y)(?=Z)** означает:

Найти X.

Проверить, идёт ли Y сразу после X (если нет – не подходит).

Проверить, идёт ли Z сразу после X (если нет – не подходит).

Если обе проверки прошли – совпадение найдено.

Негативная опережающая проверка

Допустим, нам нужно узнать из этой же строки количество индеек, то есть число \d+, за которым НЕ следует знак €. Для этой задачи мы можем применить негативную опережающую проверку.

Синтаксис: **X(?!Y)**

Он означает: найди такой X, за которым НЕ следует Y.

```
let str = "2 индейки стоят 60€";
alert( str.match(/\d+(?!€)/) ); // 2 (в этот раз проигнорирована цена)
```

Ретроспективная проверка

Опережающие проверки позволяют задавать условия на то, что «идёт после». Ретроспективная проверка выполняет такую же функцию, но с просмотром назад. Другими словами, она находит соответствие шаблону, только если перед ним есть что-то заранее определённое.

Позитивная ретроспективная проверка: **(?<=Y)X** ищет совпадение с X при условии, что перед ним ЕСТЬ Y.

Негативная ретроспективная проверка: **(?<!Y)X** ищет совпадение с X при условии, что перед ним НЕТ Y.

Скобочные группы

Как правило, то что находится внутри скобок, задающих опережающую и ретроспективную проверку, не включается в результат совпадения. Например, в шаблоне \d+(?=€) знак € не будет включён в результат. Это логично, ведь мы ищем число \d+, а (?=€) – это всего лишь проверка, что за ним идёт знак €. Но в некоторых ситуациях нам может быть интересно захватить и то, что в проверке. Для этого нужно обернуть это в дополнительные скобки.

В следующем примере знак валюты (€|kr) будет включён в результат вместе с суммой:

```
let str = "1 индейка стоит 30€";
let regexp = /\d+(?=(€|kr))/; // добавлены дополнительные скобки вокруг €|kr
alert( str.match(regexp) ); // 30, €
```

То же самое можно применить к ретроспективной проверке:

```
let str = "1 индейка стоит $30";
let regexp = /(?<=(\$|£))\d+/;
alert( str.match(regexp) ); // 30, $
```