**Лабораторная работа 1. Валидация формы.**

**Задание:**

1. **Cверстать форму по образцу на картинке (рис. 1).**

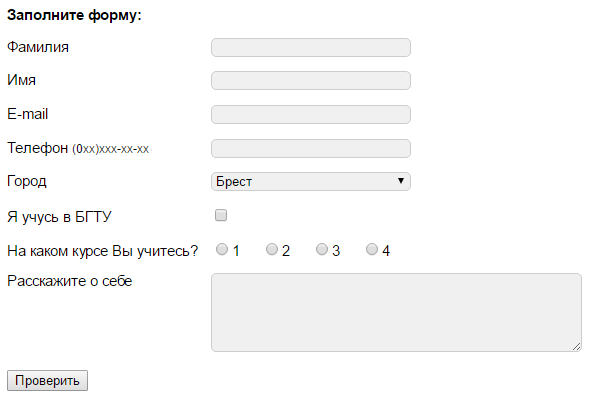


Рис. 1.

1. **На JavaScript реализовать валидацию формы. Если значение поля не соответствует правилу, необходимо сообщить об это пользователю, как показано на рис. 2.** **Проверка формы должна осуществляться по нажатию на кнопку «Проверить» типа submit. Для проверки формата и содержания полей используйте регулярные выражения.**

Правила валидации:

1. Поля «Фамилия» и «Имя»:

– размер поля не должен превышать 20 символов;

– поле должно содержать только символы русского и английского алфавита.

1. Поле «E-mail»:

– поле не должно содержать пробелы;

– после знака “@” допускается от 2 до 5 символов английского алфавита, после “.” – от 2 до 3. Допустимый формат доменного имени “@ххххх.ххх”.

1. Поле «Телефон»:

– поле должно содержать только цифры и символы “(”, “)”, “-”;

– допустимый формат поля “(0хх)ххх-хх-хх”.

1. Поле «Расскажите о себе»:

– размер поля не должен превышать 250 символов.

1. Все поля обязательны к заполнению.

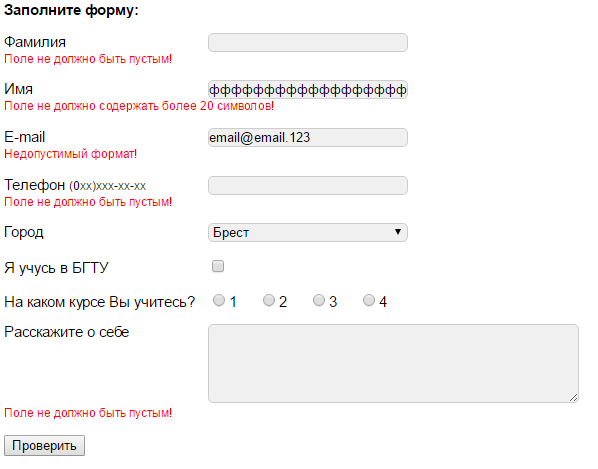


Рис. 2.

1. **Проверьте, чтобы поле «Город» содержало значение «Минск», поле «На каком курсе Вы учитесь?» – выбрано значение «3» и отмечено, что Вы учитесь в БГТУ. Если выбраны другие значения – с помощью модального окна (confirm) уточните у пользователя, уверен ли он в своем ответе. Если пользователь подтвердит свой ответ, то разрешите отправку формы на сервер, иначе – дайте ему возможность изменить значение поля.**

**Валидация формы**

Форму можно валидировать как на сервере, так и на клиенте. Когда мы валидируем форму на сервере пользователь должен дождаться ответа от сервера, валидация формы на клиенте гораздо быстрее дает пользователю понять правильно ли он ввел информацию.  Также мы можем не отправлять форму на сервер если она не проходит проверку на клиенте.

Есть несколько типов проверки, которую должны проходить поля формы - это обязательные поля и правильность формата.

Наиболее очевидная проверка – это наличие незаполненных обязательных полей, без которых данные формы не являются полезными для составителя формы. Таким образом, при валидации необходимо убедиться, что пользователь ввел данные в такие поля. Необходимо визуально обозначить такие поля как обязательные, чаще всего для этого используют звездочку, но так как не все пользователи понимает, что значит звездочка, можно добавить еще и сообщение о том что это поле является обязательным.

Не рекомендуется отмечать поля как обязательные после того как пользователь нажмет кнопку “Отправить форму” (submit). Старайтесь обходиться минимумом обязательных полей. Делайте обязательными только самые необходимые данные.

Кроме того, что нужно убедиться, что пользователь вводит **всю** необходимую информацию, важно чтобы эта информация была **в правильном формате**. Это относится к полям email, дат, чисел, номера телефона и тому подобного. Если формат данных не подходит, пользователь должен четко это понимать из сообщения об ошибке формата.

Желательно как можно более ясно дать пользователю понять в каком виде вы хотите получить от него информацию.

Нежелательно заставлять пользователя долго думать, поэтому **показывайте сообщение об ошибке формата как можно раньше**, то есть, в то время, как пользователь вводит информацию.

Если поле не проходит валидацию, ваша страница должна сообщить пользователю предельно ясно что именно он сделал не так и в каком виде вы хотите получить от него информацию. Пользователь должен легко заметить ваше сообщение об ошибке таким образом, чтобы у него не возникало вопросов почему форма не отправляется.

Желательно выделять сообщение об ошибке красным цветом текста или выделять поле, в котором пользователь ошибся красной рамкой для того, чтобы привлечь его внимание.

Можно использовать понятную иконку красного цвета, которая должна помочь определить ошибку.

Это плохо и непонятно выглядит, когда вы перечисляете все ошибки в одном поле сверху даже если это большой красный список, особенно если этот красный список появляется после нажатия кнопки отправить форму, которая еще и стирает все ранее введенные данные в форму.

Пользователю гораздо проще стереть несколько введенных символов чем перезаполнять форму сначала.

Предпочтительной будет проверка формата данных непосредственно во время ввода.

Позаботьтесь о том, чтобы сообщения об ошибках исчезали также моментально, когда пользователь вводит правильную информацию, чтобы не запутать пользователя когда он не ошибается.

Если вы используете Javascript для валидации, не забывайте добавлять код, который убирает сообщения об ошибках.

Будет полезно если вы поможете пользователю ввести правильные данные короткими и понятными подсказками возле полей формы. Предпочтительнее будут сообщения которые показываются всегда или в тот момент когда пользователь начинает ввод в поле, то есть фокусируется на поле. Менее удобными для пользователя будут всплывающие подсказки которые показываются на наведении мышки.

**Регулярные выражения**

Проверить правильный формат и содержание поля вам помогут регулярные выражения.

**Регулярное выражение** (regular expression, regexp, регэксп) - механизм, позволяющий задать шаблон для строки и осуществить поиск данных, соответствующих этому шаблону в заданном тексте. Кроме того, дополнительные функции по работе с regexp'ами позволяют получить найденные данные в виде массива строк, произвести замену в тексте по шаблону, разбиение строки по шаблону и т.п. Однако главной их функцией, на которой основаны все остальные, является именно функция поиска в тексте данных, соответствующих шаблону, описанному в синтаксисе регулярных выражений.

Очень часто регулярные выражения используются для того, чтобы проверить, является ли данная строка строкой в необходимом формате. Например следующий regexp предназначен для проверки того, что строка содержит корректный e-mail адрес:

/^\w+([\.\w]+)\*\w@\w((\.\w)\*\w+)\*\.\w{2,3}$/

Выглядит, на первый взгляд, довольно сложно. Но, тем не менее, это работает, и работает очень хорошо. А когда вы научитесь писать и использовать regexp'ы в своем коде - это еще будет и сильно облегчать вам жизнь.

Сутью механизма регулярных выражений является то, что они позволяют задать шаблон для нечеткого поиска по тексту. Например, если перед вами стоит задача найти в тексте определенное слово, то с этой задачай хорошо справляются и обычные функции работы со строками. Однако если вам нужно найти "то, не знаю что", о чем вы можете сказать только то, как приблизительно это должно выглядеть - то здесь без регулярных выражений просто не обойтись. Например, вам необходимо найти в тексте информацию, про которую вам известно только то, что это "3 или 4 цифры после которых через пробел идет 5 заглавных латинских букв", то вы сможете сделать это очень просто, возпользовавшись следующим регулярным выражением:

/\d{3,4}\s[A-Z]{5}/

Синтаксис регулярных выражений

Регулярные выражения, как уже было сказано выше, представляют собой строку. Строка всегда начинается с символа разделителя, за которым следует непосредственно регулярное выражение, затем еще один символ разделителя и потом необязятельный список модификаторов. В качестве символа разделителя обычно используется слэш ('/'). Таким образом в следующем регулярном выражении: /\d{3}-\d{2}/m, символ '/' является разделителем, строка '\d{3}-\d{2}' - непосредственно регулярным выражением, а символ 'm', расположенный после второго разделителя - это модификатор.

Основой синтаксиса регулярных выражений является тот факт, что некоторые символы, встречающиеся в строке рассматриваются не как обычные символы, а как имеющие специальное значение (т.н. метасимволы). Именно это решение позволяет работать всему механизму регулярных выражений. Каждый метасимвол имеет свою собственную роль в синтаксисе регулярных выражений. Далее мы рассмотрим все эти метасимволы.

Одним из самых важных метасимволов является символ обратного слэша ('\'). Если в строке встречается этот символ, то парсер рассматривает символ, непосредственно следующий за ним двояко:

* если следующий символ в обычном режиме имеет какое-либо специальное значение, то он теряет это свое специальное значение и рассматривается как обычный символ. Это совершенно необходимо для того, чтобы иметь возможность вставлять в строку специальные символы, как обычные. Например метасимвол '.', в обычном режиме означает "любой единичный символ", а '\.' означает просто точку. Также можно лишить специального значения и сам этот символ: '\\'.
* если следующий символ в обычном режиме не имеет никакого специального значения, то он может получить такое значение, будучи соединенным с символом '\'. К примеру символ 'd' в обычном режиме воспринимается просто как буква, однако, будучи соединенной с обратным слэшем ('\d') становится метасимволом, означающим "любая цифра".

Существует множество символов, которые образуют метасимволы в паре с обратным слэшем. Как правило подобные пары используются для того, чтобы показать, что на этом месте в строке должен находиться символ, с кодом, который не имеет соответствующего ему изображения или же символ, принадлежащий какой-то определенной группе символов. Ниже приведены некоторые наиболее употребительные:

|  |  |
| --- | --- |
| **Метасимвол** | **Значение** |
| Метасимволы для задания символов, не имеющих изображения | |
| \n | Символ перевода строки (код 0x0A) |
| \r | Символ возврата каретки (код 0x0D) |
| \t | Символ табуляции (код 0x09) |
| \xhh | Вставка символа с шестнадцатиричным кодом 0xhh, например \x41 вставит латинскую букву 'A' |
| Метасимволы для задания групп символов | |
| \d | Цифра (0-9) |
| \D | Не цифра (любой символ кроме символов 0-9) |
| \s | Пустой символ (обычно пробел и символ табуляции) |
| \S | Непустой символ (все, кроме символов, определяемых метасимволом \s) |
| \w | "Словесный" символ (символ, который используется в словах. Обычно все буквы, все цифры и знак подчеркивания ('\_')) |
| \W | Все, кроме символов, определяемых метасимволом \w |

В таблице ниже несколько простейших примеров. Примеры несколько громоздки и некрасивы, потому, что не использованы в них метасимволы, которые сделали бы их намного проще.

|  |  |
| --- | --- |
| **Regexp** | **Комментарии** |
| /\d\d\d/ | Любое трехзначное число ('123', '719', '001') |
| /\w\s\d\d/ | Буква, пробел (или табуляция) и двузначное число ('A 01', 'z 45', 'S 18') |
| /\d and \d/ | Любая из следующих строк: '1 and 2', '9 and 5', '3 and 4'. |

Синтаксис регулярных выражений имеет средства для определения собственных подмножеств символов. Например, вам может понадобиться задать условие, что в этом месте строки должна находиться шестнадцатиричная цифра или еще что-то подобное. Для описания таких подмножеств применяются символы квадратных скобок '[]'. Квадратные скобки, встреченные внутри регулярного выражения считаются одним символом, который может принимать значения, перечисленные внутри этих скобок.

Есть небольшая тонкость в том, как работают метасимволы внутри квадратных скобок. Дело в том, что в синтаксисе регулярных выражений существует еще множество метасимволов, но практически все они работают только вне секций описаний подмножеств. Единственные метасимволы, которые работают внутри этих секций это:

* Обратный слэш ('\'). Т.е. все метасимволы из приведенной ранее таблицы будут работать.
* Минус ('-'). Используется для задания набора символов из одного промежутка (например все цифры могут быть заданы как '0-9')
* Символ '^'. Если этот символ стоит первым в секции задания подмножества символов (и только в этом случае!) он будет рассматриваться как символ отрицания. Т.о. можно задать все символы, которые не описаны в данной секции.

Несколько примеров, чтобы было понятно, как это работает:

|  |  |
| --- | --- |
| **Regexp** | **Комментарии** |
| [0-9A-Fa-f] | Цифра в шестнадцатиричной системе счисления |
| [\dA-Fa-f] | То же самое, но с использованием метасимвола |
| [02468] | Четная цифра |
| [^\d] | Все, кроме цифр (аналог метасимвола \D) |
| [a^b] | Любой из символов 'a', 'b', '^'. Заметьте, что здесь символ '^' не имеет какого-либо специального значения, потому что стоит не на первой позиции внутри квадратных скобок. |

Теперь необходимо рассмотреть еще несколько метасимволов. Как уже было сказано ранее, все они работают только вне секций описаний подмножеств символов (вне квадратных скобок).

Символы '^' и '$'. Они использутся для того, что того, чтобы указать парсеру регулярных выражений на то, чтобы он обратил внимание на положение искомого текста в строке. Символ '^' указывает, что искомый текст должен находиться в начале строки, символ '$' наоборот, указывает, что искомый текст должен находиться в конце строки. Посмотрим, как это работает на примере:

Допустим, у нас есть текст:

12 aaa bbb

aaa 27 ccc

aaa aaa 45

И регулярное выражение для поиска чисел в этом тексте: /\d\d/m (не обращайте пока внимания на модификатор). Поиск по этому регулярному выражению вернет нам 3 значения: '12', '27', '45'. Теперь ограничим поиск, указав, где именно внутри строки должен располагаться текст: /^\d\d/m. Здесь результат будет только один - '12', потому что только это число располагается в начале строки. Аналогично, регулярное выражение /\d\d$/m вернет результат '45'.

Символ точки '.'. Этот метасимвол указывает, что на данном месте в строке может находиться любой символ (за исключением символа перевода строки). Очень удобно использовать его, если вам нужно "пропустить" какую-нибудь букву в слове при проверке. Например регулярное выражение /.bc/ найдет в тексте и 'abc' и 'Abc' и 'Zbc' и '5bc'.

Символ вертикальной черты '|'. Используется для задания списка альтернатив. Например регулярное выражение:

/(красное зеленое) яблоко/

Найдет в тексте все словосочетания 'красное яблоко' и 'зеленое яблоко'. О значении круглых скобок в этом выражении см. далее.

Символы круглых скобок '(' и ')'. Эти символы позволяют получить из искомой строки дополнительную информацию. Обычно, если парсер регулярных выражений ищет в тексте информацию по заданному выражению и находит ее - он просто возвращает найденную строку. Однако, если он встречает внутри регулярного выражения круглые скобки, то он рассматривает содержимое этих скобок как еще одно регулярное выражение, по которому необходимо произвести поиск. Парсер рекурсивно вызывает сам себя для поиска по новому регулярному выражению и использует результаты поиска для дальнейшей обработки основного регулярного выражения. При этом, если поиск хотя бы по одному из внутренних регулярных выражений не увенчался успехом - поиск по всему регулярному выражению считается безуспешным.

Рассмотрим в качестве примера то, как работает парсер регулярных выражений в случае приведенного выше регулярного выражения о яблоках: /(красное зеленое) яблоко/.

1. Парсер начинает разбор регулярного выражения и встречает выражение в скобках: (красное зеленое)
2. Парсер вызывает себя для поиска по найденному регулярному выражению
3. Получив результаты поиска парсер подставляет по очереди каждый из полученных результатов на место выражения в скобках и смотрит, удовлетворяет ли найденный результат всем условиям основного регулярного выражения (в данном случае смотрит, есть ли после найденного слова слово "яблоко").
4. Если все в порядке - результаты поиска по каждому из имеющихся регулярных выражений для этого случая возвращаются, если нет - парсер просто переходит к следующему найденному фрагменту. Результат поиска внутреннего регулярного выражения для этого фрагмента при этом теряется.

В качестве примера возьмем строку:

яблоко красное и зеленое яблоко и еще одно красное яблоко и еще одно яблоко, зеленое

Поиск по внутреннему регулярному выражению даст 4 результата (выделены жирным шрифтом):

яблоко красное и зеленое яблоко и еще одно красное яблоко и еще одно яблоко, зеленое

Однако поиск по всему регулярному выражению даст всего 2 результата, потому как в остальных случаях условия основного регулярного выражения не выполняются:

яблоко красное и зеленое яблоко и еще одно красное яблоко и еще одно яблоко, зеленое

Необходимо заметить, что для этих двух случаев будет возвращен не только результат поиска по основному регулярному выражению, но и результат поиска по внутреннему регулярному выражению для каждого из найденных фрагментов. В большинстве случаев это полезно (пример - чуть позднее), но иногда наоборот, лучше избавиться от лишних результатов. В этом случае необходимо добавить символы '?:' непосредственно после открывающейся круглой скобки: /(?:красное зеленое) яблоко/.

Теперь пример, когда получение результатов внутренних регулярных выражений может быть полезным. Допустим, нам необходимо проверить, является ли строка семизначным телефонным номером с указанием кода города и получить из нее код города и номер телефона:

/\((\d{3,5})\)\s+(\d{3}-\d{2}-\d{2})/

Некоторые из примененных здесь метасимволов вам еще неизвестны и будут рассмотрены чуть позднее. Давайте рассотрим этот regexp подробнее.

Первая круглая скобка здесь теряет свое специальное значение и будет рассматриваться как обычный символ:

/\((\d{3,5})\)\s+(\d{3}-\d{2}-\d{2})/

Далее идет регулярное выражение в скобках (проверка кода города):

/\((\d{3,5})\)\s+(\d{3}-\d{2}-\d{2})/

После этого идет закрывающая круглая скобка, которая также лишена своего специального значения из-за символа обратного слэша, стоящего перед ней:

/\((\d{3,5})\)\s+(\d{3}-\d{2}-\d{2})/

Затем идет пропуск пустого места:

/\((\d{3,5})\)\s+(\d{3}-\d{2}-\d{2})/

И еще одно регулярное выражение в скобках, которое проверяет номер телефона:

/\((\d{3,5})\)\s+(\d{3}-\d{2}-\d{2})/

Видно, что здесь есть 3 регулярных выражения - основное и два внутренних. При этом основное выражение позволяет проверить, имеет ли строка необходимый формат, а два внутренних - получить соответственно код города и номер телефона. Т.е. одним регулярным выражением можно решить сразу несколько задач!

Посмотрим, как работает это регулярное выражение. Пусть у нас есть строка: "My phone is (095) 123-45-67". Результатами поиска будут 3 строки: '(095) 123-45-67', '095' и '123-45-67'.

Еще одна группа метасимволов, определяющих количественные показатели (т.н. ***quantifiers, “кванторы”***). Очень часто бывает необходимо указать, что какой-то символ должен повторяться определенное количество раз. Конечно, можно просто указать его необходимое количество раз непосредственно в строке, но это, естественно не выход. Тем более, что очень часто встречаются ситуации, когда точное количество символов неизвестно. Поэтому синтаксис регулярных выражений содержит набор метасимволов, предназначенных именно для решения подобных задач. Каждый из описанных ниже метасимволов определяет количественную характеристику символа который находится **непосредственно** перед ним.

Звездочка '\*'. Указывает, что символ должен быть повторен 0 или более раз (т.е. символ может отсутствовать или присутствовать в любых количествах). Пример: выражение /ab\*c/ найдет строки 'ac', 'abc', 'abbc' и т.д.

Плюс '+'. Указывает, что символ должен быть повторен 1 или более раз (т.е. символ обязан присутствовать и может присутствовать в любых количествах). Пример: выражение /ab+c/ найдет строки 'abc', 'abbc', 'abbbc' и т.д., но не найдет строку 'ac'.

Знак вопроса '?'. Указывает, что символ моет как присутствовать, так и нет, но при этом не может повторяться более одного раза. Пример: выражение /ab?c/ найдет строки 'ac' и 'abc', но не найдет строку 'abbc'.

Фигурные скобки '{' и '}'. Определяют количественную характеристику символа. Внутри скобок через запятую перечисляются минимальное и максимальное количество повторений символа. При этом любой из параметров может быть опущен, а кроме того можно задать точное количество повторений, указав только одно число. Примеры:

* {2,4} - символ долен повториться минимум 2 раза, но не более 4.
* {,5} - символ может отсутствовать (т.к. не задано минимальное количество повторений), но если присутствует, то не должен повторяться более 5 раз.
* {3,} - символ должен повторяться минимум 3 раза, но может быть и больше.
* {4} - символ должен повторяться ровно 4 раза

Есть еще одна тонкость в использовании метасимвола '?'. Посмотрите на такое выражение: /.+a/. Ожидается, что оно вернет часть текста до первого вхождения символа 'a' в этот текст. На самом деле оно будет работать несколько не так, как ожидается и результатом поиска будет весь текст до последнего вхождения символа 'a'. Дело в том, что по умолчанию количественные метасимволы "жадничают" и пытаются захватить как можно больший кусок текста. Если это не нужно (как а данном случае), то необходимо "отучить" их от жадности, указав знак '?' после количественного метасимвола: /.+?a/. После этого выражение будет работать так как надо.

**Модификаторы регулярных выражений**

Как уже было сказано ранее - механизм регулярных выражений позволяет добавлять модификаторы, влияющие на обработку регулярного выражения. Ниже рассмотрены наиболее употребительные.

|  |  |
| --- | --- |
| **Модификатор** | **Значение** |
| i | Включение режима case-insensitive, т.е. большие и маленькие буквы в выражении не различаются. |
| m | Указывает на то, что текст, по которому ведется поиск, должен рассматриваться как состоящий из нескольких строк. По умолчанию механизм регулярных выражений рассматривает текст как одну строку вне зависимости от того, чем она является на самом деле. Соответственно метасимволы '^' и '$' указывают на начало и конец всего текста. Если же этот модификатор указан, то они будут указывать соответственно на начало и конец каждой строки текста. |
| s | По умолчанию метасимвол '.' не включает в свое определение символ перевода строки. Т.е. для многострочного текста выражение /.+/ вернет только первую строку, а не весь текст, как ожидается. Указание этого модификатора снимает это ограничение. |
| U | Делает все количественные метасимволы "не жадными" по умолчанию (про "жадность" количественных метасимволов см. выше) |

**Спецсимволы в регулярном выражении**

|  |  |
| --- | --- |
| **Символ** | **Значение** |
| \ | Для обычных символов - делает их специальными. Например, выражение /s/ ищет просто символ 's'. А если поставить \ перед s, то /\s/ уже обозначает пробельный символ. И наоборот, если символ специальный, например \*, то \ сделает его просто обычным символом "звездочка". Например, /a\*/ ищет 0 или больше подряд идущих символов 'a'. Чтобы найти а со звездочкой 'a\*' - поставим \ перед спец. символом: /a\\*/. |
| ^ | Обозначает начало входных данных. Если установлен флаг многострочного поиска ("m"), то также сработает при начале новой строки. Например, /^A/ не найдет 'A' в "an A", но найдет первое 'A' в "An A". |
| $ | Обозначает конец входных данных. Если установлен флаг многострочного поиска, то также сработает в конце строки. Например, /t$/ не найдет 't' в "eater", но найдет - в "eat". |
| \* | Обозначает повторение 0 или более раз. Например, /bo\*/ найдет 'boooo' в "A ghost booooed" и 'b' в "A bird warbled", но ничего не найдет в "A goat grunted". |
| + | Обозначает повторение 1 или более раз. Эквивалентно {1,}. Например, /a+/ найдет 'a' в "candy" и все 'a' в "caaaaaaandy". |
| ? | Обозначает, что элемент может как присутствовать, так и отсутствовать. Например, /e?le?/ найдет 'el' в "angel" и 'le' в "angle". Если используется сразу после одного из квантификаторов \*, +, ?, или {}, то задает "нежадный" поиск (повторение минимально возможное количество раз, до ближайшего следующего элемента паттерна), в противоположность "жадному" режиму по умолчанию, при котором количество повторений максимально, даже если следующий элемент паттерна тоже подходит. Кроме того, ? используется в предпросмотре, который описан в таблице под (?=), (?!), и (?: ). |
| . | (Десятичная точка) обозначает любой символ, кроме перевода строки: \n \r \u2028 or \u2029. (можно использовать [\s\S] для поиска любого символа, включая переводы строк). Например, /.n/ найдет 'an' и 'on' в "nay, an apple is on the tree", но не 'nay'. |
| (x) | Находит x и запоминает. Это называется "запоминающие скобки". Например, /(foo)/ найдет и запомнит 'foo' в "foo bar". Найденная подстрока хранится в массиве-результате поиска или в предопределенных свойствах объекта RegExp: $1, ..., $9.Кроме того, скобки объединяют то, что в них находится, в единый элемент паттерна. Например, (abc)\* - повторение abc 0 и более раз. |
| (?:x) | Находит x, но не запоминает найденное. Это называется "незапоминающие скобки". Найденная подстрока не сохраняется в массиве результатов и свойствах RegExp. Как и все скобки, объединяют находящееся в них в единый подпаттерн. |
| x(?=y) | Находит x, только если за x следует y. Например, /Jack(?=Sprat)/ найдет 'Jack', только если за ним следует 'Sprat'. /Jack(?=Sprat|Frost)/ найдет 'Jack', только если за ним следует 'Sprat' или 'Frost'. Однако, ни 'Sprat' ни 'Frost' не войдут в результат поиска. |
| x(?!y) | Находит x, только если за x не следует y. Например, /\d+(?!\.)/ найдет число, только если за ним не следует десятичная точка. /\d+(?!\.)/.exec("3.141") найдет 141, но не 3.141. |
| x|y | Находит x или y. Например, /green|red/ найдет 'green' в "green apple" и 'red' в "red apple". |
| {n} | Где n - положительное целое число. Находит ровно n повторений предшествующего элемента. Например, /a{2}/ не найдет 'a' в "candy", но найдет оба a в "caandy", и первые два a в "caaandy". |
| {n,} | Где n - положительное целое число. Находит n и более повторений элемента. Например, /a{2,} не найдет 'a' в "candy", но найдет все 'a' в "caandy" и в "caaaaaaandy". |
| {n,m} | Где n и m - положительные целые числа. Находят от n до m повторений элемента. |
| [xyz] | Набор символов. Находит любой из перечисленных символов. Вы можете указать промежуток, используя тире. Например, [abcd] - то же самое, что [a-d]. Найдет 'b' в "brisket" и 'c' в "ache". |
| [^xyz] | Любой символ, кроме указанных в наборе. Вы также можете указать промежуток. Например, [^abc] - то же самое, что [^a-c]. Найдет 'r' в "brisket" и 'h' в "chop". |
| [\b] | Находит символ backspace. (Не путать с \b.) |
| \b | Находит границу слов (латинских), например пробел. (Не путать с [\b]). Например, /\bn\w/ найдет 'no' в "noonday"; /\wy\b/ найдет 'ly' в "possibly yesterday". |
| \B | Обозначает не границу слов. Например, /\w\Bn/ найдет 'on' в "noonday", а /y\B\w/ найдет 'ye' в "possibly yesterday". |
| \cX | Где X - буква от A до Z. Обозначает контрольный символ в строке. Например, /\cM/ обозначает символ Ctrl-M. |
| \d | Находит цифру из любого алфавита (юникод). Используйте [0-9], чтобы найти только обычные цифры. Например, /\d/ или /[0-9]/ найдет '2' в "B2 is the suite number". |
| \D | Найдет нецифровой символ (все алфавиты). [^0-9] - эквивалент для обычных цифр. Например, /\D/ или /[^0-9]/ найдет 'B' в "B2 is the suite number". |
| \f,\r,\n | Соответствующие спецсимволы form-feed, line-feed, перевод строки. |
| \s | Найдет любой пробельный символ, включая пробел, табуляцию, переводы строки и другие юникодные пробельные символы. Например, /\s\w\*/ найдет ' bar' в "foo bar". |
| \S | Найдет любой символ, кроме пробельного. Например, /\S\w\*/ найдет 'foo' в "foo bar". |
| \t | Символ табуляции. |
| \v | Символ вертикальной табуляции. |
| \w | Найдет любой словесный (латинский алфавит) символ, включая буквы, цифры и знак подчеркивания. Эквивалентно [A-Za-z0-9\_]. Например, /\w/ найдет 'a' в "apple", '5' в "$5.28" и '3' в "3D". |
| \W | Найдет любой не-(лат.) словесный символ. Эквивалентно [^A-Za-z0-9\_]. Например, /\W/ и /[^$A-Za-z0-9\_]/ одинаково найдут '%' в "50%". |
| \n | Где n - целое число. Обратная ссылка на n-ю запомненную скобками подстроку. Например, /apple(,)\sorange\1/ найдет 'apple, orange,' в "apple, orange, cherry, peach.". За таблицей есть более полный пример. |
| \0 | Найдет символ NUL. Не добавляйте в конец другие цифры. |
| \xhh | Найдет символ с кодом hh (2 шестнадцатиричных цифры) |
| \uhhhh | Найдет символ с кодом hhhh (4 шестнадцатиричных цифры). |

**Операторы для работы с регулярными выражениями в Javascript**

Регулярные выражения используются методами exec и test объекта RegExp и методами match, replace, search и split объекта String. Если нам нужно просто проверить, содержит ли данная строка подстроку, соответствующую образцу, то используются методы test или search. Если же нам необходимо извлечь подстроку (или подстроки), соответствующие образцу, то нам придется воспользоваться методами exec или match. Метод replace обеспечивает поиск заданной подстроки и замены ее на другую строку, а метод split позволяет разбить строку на несколько подстрок, основываясь на регулярном выражении или обычной текстовой строке.

**exec(regexp)** - находит все совпадения (вхождения в шаблон "регулярки") в строке. Возвращает массив (при совпадении) и обновляет свойство regexp-а, или null - если ничего не найдено,. С модификатором g - при каждом вызове этой функции, она будет возвращать следующее совпадение после предыдущего найденного - это реализовано с помощью ведения индекса смещения последнего поиска.

**match(regexp)** - найти часть строки по шаблону. Если указан модификатор g, то функция match() возвращает массив всех совпадений или null (а не пустой массив). Без модификатора g эта функция работает как exec();

**replace(regexp, mix)** - метод возвращает строку измененную в соответствии с шаблоном (регулярным выражением). Первый параметр regexp также может содержать строку, а не регулярное выражение. Без модификатора g - метод в строке заменяет только первое вхождение; с модификатором g - происходит глобальная замена, т.е. меняются все вхождения в данной строке. mix - шаблон замены, может принимать значения строки, шаблона замены, функции (имя функции).

**test(regexp)** - функция проверяет строку на соответствие шаблону. Возвращает true - если есть совпадение, и false - если совпадения нет.

Метод test проверяет, есть ли хоть одно совпадение в строке str. Работает, по сути, так же, как и проверка str.search(reg) != -1, например:

var str = "Любовь";

// эти две проверки идентичны

alert( /лю/i.test(str) ) // true

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