Урок 4

Регулярные выражения

Регулярные выражения в JavaScript. Основные операции с регулярными выражениями и строками.

Часто в программе нам нужно проверить, соответствует ли строка заданному формату. Представьте, что пользователь заполняет форму. Он вводит телефон и адрес электронной почты. Перед отправкой формы нужно проверить, правильно ли пользователь заполнил поля – например, не попали ли случайно в номер телефона буквы. Другая ситуация: нам нужно заменить в тексте одни символы на другие – например, двойные кавычки на одинарные. Для решения подобных задач используются регулярные выражения.

Что такое регулярное выражение

Регулярное выражение – способ записи текстовых шаблонов. По сути, это обычная текстовая строка написанная на специальном языке, но она не содержит конкретный набор символов, а задаёт общие правила. Например, такие: строка из 5 символов, первый символ – Q, последний – цифра.

В JavaScript регулярное выражение можно объявить двумя способами:

```
const option1 = new RegExp('pattern');  // Создание через конструктор
const option2 = /pattern/;              // Создание синтаксическим способом
```

Вот несколько основных правил языка регулярных выражений:

```
/abc/            Идущие подряд символы abc
/[abc]/          Один из символов a, b или c
/[^abc]/         Ни один из символов, т. е. не a, не b и не c
/[a-z]/          Диапазон символов, идущих подряд в таблице Unicode
/\b/             Граница слова
/\B/             Не граница слова
/\d/             Цифра
/\D/             Не цифра
/\w/             Латинская буква, цифра или _
/\W/             Не латинская буква, не цифра и не _
/\s/             Пробельный символ
/\S/             Непробельный символ
/a{3}/           Строго 3 символа a подряд
/a{2,4}/         От 2 до 4 символов a подряд
/a+/             1 и более символов a подряд
/a*/             0 и более символов a подряд
/a?/             0 или 1 символ a
/./              Один любой символ, кроме переноса строки
```

На самом деле правил значительно больше, но запоминать их не нужно. Всегда можно обратиться к справочнику. Комбинируя правила, можно создавать шаблоны:

```
/a25?0+/         Шаблон для строки, которая начинается с a2, затем
                 в ней может быть цифра 5, а потом один и более нулей.
                 То есть подойдут строки a20, a250, a2000000 и т. д.
```

Если в шаблоне нужно использовать служебные символы вроде **?** или **+**, то их нужно экранировать:

```
/abc\?/          Найдёт 'abc?'
```

Также в регулярных выражениях есть флаги, влияющие на поиск. Вот три основных:

```
g   // global           Глобальный поиск, т. е. поиск всех соответствий
i   // insensitive      Поиск без учёта регистра
m   // multiline        Многострочный текст
```

Флаг устанавливается за закрывающим слешем или вторым параметром конструктора **RegExp()**:

```
const regexp1 = new RegExp('abc', 'g');
const regexp2 = /abc/g;
```

Флаги можно комбинировать в одном шаблоне:

```
/abc/              // Будет искать первое совпадение
/abc/g             // Будет искать все совпадения
/abc/gi            // Будет искать все совпадения без учёта регистра
```

Чтобы добавить в шаблон логическое ИЛИ, используют символ |:

```
/gr(a|e)y/         // Найдёт 'gray' и 'grey'
```

В данном случае скобки ограничивают область действия логического ИЛИ. Чуть позже рассмотрим, для чего ещё их можно применять.

Как узнать, есть ли совпадение

Для того чтобы определить, есть ли в строке совпадение с регулярным выражением, используется метод регулярного выражения **test()**. Он не определяет, где именно обнаружено совпадение, а лишь возвращает **true** или **false**:

```
const str = '123 abc 456';

var regexp = /abc/;
regexp.test(str);  // Вернёт true

var regexp2 = /xyz/;
regexp2.test(str); // Вернёт false
```

Как получить совпадения

Чтобы найти совпадения, используют метод строки **match()**. Этот метод возвращает массив найденных совпадений:

```
const str = 'Geek from Geekbrains';
const regexp = /Geek/;
```

```
console.log(str.match(regexp));    // Вернёт ['Geek']
```

Если указать у регулярного выражения флаг **g**, то в массиве будут возвращены все совпадения:

```
const str = 'Geek from Geekbrains';
const regexp = /Geek/g;

console.log(str.match(regexp));    // Вернёт ['Geek', 'Geek']
```

Как заменить часть строки на другую

Для замены подстроки по шаблону используется метод строки **replace**:

```
const str = 'This is string';
const regexp = /is/g;               // Ищет все вхождения is

str.replace(regexp, '+');           // Заменит is на + и вернёт 'Th+ + string'
```

С помощью скобок части шаблона можно объединять в группы:

```
const str = 'Hi, I am Greek geek from Geekbrains';
const regexp = /(g.+?k)/gi;

str.replace(regexp, '+');             // Вернёт "Hi, I am + + from +brains"
```

В строке, которую мы подставляем вместо найденной по шаблону, можно обратиться к содержимому группы. Для этого используется символ **$** и порядковый номер группы, если считать слева направо.

```
const str = 'Hi, I am Greek geek from Geekbrains';
const regexp = /(g.+?k)/gi;

str.replace(regexp, '+$1+');          // Вернёт "Hi, I am +Greek+ +geek+ from
                                      // +Geek+brains"
```

Что такое «жадный» поиск

Возьмём такую строку:

```
const str = '000 1221 133331';
```

Предположим, что мы хотим найти все подстроки, которые начинаются и кончаются цифрой 1, а между ними любое количество символов. Напишем регулярное выражение и найдём все вхождения с помощью **match**.

```
var regexp = /1.+1/g;
var text = '000 1221 133331';
```

```
text.match(regexp);
```

Мы ожидаем, что нам вернётся ['1221', '133331'], но на деле результат будет ['1221 133331']. Это называется «жадный» поиск: будет найдена подстрока максимальной длины, удовлетворяющая условию. Для того, чтобы сделать поиск «нежадным», или «ленивым» (т. е. найти минимальную подстроку, удовлетворяющую условию), нужно добавить символ **?** перед символом **+**:

```
var regexp = /1.+?1/g;
```

Теперь регулярное выражение будет работать так, как мы изначально задумывали, и вернёт ['1221', '133331']. Это правило распространяется и на другие команды, которые определяют количество повторений символа или группы, например * и **{}**.

Полезные регулярные выражения

Идеальных регулярных выражений не существует. Для каждого шаблона можно подобрать пример, который этот шаблон не охватывает. Несмотря на это, есть несколько проверенных регулярных выражений, которые закрывают почти 100% ежедневных потребностей разработчика.

URL. Учитывает протоколы http и https, дефисы, подчеркивания, точки и внутреннюю файловую структуру:

```
/^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]*)*\/?$/
```

Email. Учитывает точки и дефисы в логине и домене. Есть ограничение по длине домена первого уровня:

```
/^([a-z0-9_\.-]+)@([a-z0-9_\.-]+)\.([a-z\.]{2,6})$/
```

Практика

Сделаем для нашего списка товаров регистронезависимый поиск. Добавим в шапку поле ввода и кнопку «Искать»:

```
...
<header>
  <input type="text" class="goods-search" />
  <button class="search-button" type="button">Искать</button>
  <button class="cart-button" type="button">Корзина</button>
</header>
...
```

Добавим обработчик клика на кнопку и метод **filterGoods()** в класс **GoodsList**. Будем брать значение из поля ввода и фильтровать список:

```
class GoodsList {
```

```
  ...
  filterGoods(value) {
    // Здесь будем фильтровать список товаров
  }
  ...
}

searchButton.addEventListener('click', (e) => {
  const value = searchInput.value;
  list.filterGoods(value);
});
```

Полученный от сервера список товаров будем также хранить в поле **goods**. Дополнительно заведём новый массив в поле **filteredGoods**. Его будем использовать для отрисовки в методе **render()**. Чтобы каждый раз не запрашивать список товаров с сервера, просто будем подменять исходный список.
При получении списка товаров будем записывать его ещё и в **filteredGoods**.

```
class GoodsList {
  constructor() {
    this.goods = [];
    this.filteredGoods = [];
  }
  fetchGoods(cb) {
    makeGETRequest(`${API_URL}/catalogData.json`, (goods) => {
      this.goods = JSON.parse(goods);
      this.filteredGoods = JSON.parse(goods);
      cb();
    })
  }
  ...
}
```

Теперь обновим метод **render()**, чтобы товары отрисовывались из списка **filteredGoods**:

```
class GoodsList {
  ...
  render() {
    let listHtml = '';
    this.filteredGoods.forEach(good => {
      const goodItem = new GoodsItem(good.product_name, good.price);
      listHtml += goodItem.render();
    });
    document.querySelector('.goods-list').innerHTML = listHtml;
  }
}
```

Теперь пропишем метод **filterGoods()**. На вход получаем текстовую строку, делаем на её основе регистронезависимое регулярное выражение и фильтруем массив с помощью функции **test()**. После того, как список сформирован, запускаем метод **render()**:

```
class GoodsList {
  ...
  filterGoods(value) {
    const regexp = new RegExp(value, 'i');
    this.filteredGoods = this.goods.filter(good =>
regexp.test(good.product_name));
    this.render();
  }
  ...
}
```

Практическое задание

1. Дан большой текст, в котором для оформления прямой речи используются одинарные кавычки. Придумать шаблон, который заменяет одинарные кавычки на двойные.
2. Улучшить шаблон так, чтобы в конструкциях типа **aren't** одинарная кавычка не заменялась на двойную.
3. * Создать форму обратной связи с полями: **Имя**, **Телефон**, **E-mail**, текст, кнопка **Отправить**. При нажатии на кнопку **Отправить** произвести валидацию полей следующим образом:
   a. Имя содержит только буквы.
   b. Телефон имеет вид +7(000)000-0000.
   c. E-mail имеет вид mymail@mail.ru, или my.mail@mail.ru, или my-mail@mail.ru.
   d. Текст произвольный.
   e. Если одно из полей не прошло валидацию, необходимо выделить это поле красной рамкой и сообщить пользователю об ошибке.

Дополнительные материалы

1. В поисках идеального регулярного выражения для валидации URL.
2. Регулярные выражения.

Используемая литература

1. Современный учебник JavaScript.