

REGEXP, MATH, ARRAYBUFFER

ЛЕКТОР / КОМПАНИЯ

ЛЕКТОР

front-end developer в



ПЛАН ЗАНЯТИЯ

- [RegExp](#)
- [Math](#)
- [ArrayBuffer](#)
- [Интересное чтение](#)

REGEXP



REGEXP

При поиске "regex" Яндекс намекает на какие-то *регулярные выражения*:

Яндекс regex ✕ Найти 🔊 ⚙️

Поиск [Картинки](#) [Видео](#) [Карты](#) [Маркет](#) [Новости](#) [ТВ онлайн](#) [Знатоки](#) [Коллекции](#) [Ещё](#)

Регулярные выражения — Википедия
ru.wikipedia.org > Регулярные выражения ▾ 📄
Регулярные выражения (англ. **regular expressions**) — формальный язык поиска и осуществления манипуляций с подстроками в тексте... [Читать ещё >](#)

Шпаргалка по регулярным выражениям
website-lab.ru > article/regex/shpargalka_po... ▾
Регулярные выражения. JavaScript. **RegExp**. [Читать ещё >](#)

RegExp | MDN | Веб-документация MDN
developer.mozilla.org > Технологии > .../Global_Objects/RegExp ▾ 👑
Конструктор **RegExp** создаёт объект регулярного выражения для сопоставления текста с шаблоном. [Читать ещё >](#)

Регулярные выражения для новичков | Статьи
tproger.ru > articles/regex-for-beginners/ ▾
Англоязычное название этого инструмента — **Regular Expressions** или просто **RegExp**. Строго говоря, регулярные выражения — специальный язык для описания шаблонов строк. Реализация этого инструмента различается в разных... [Читать ещё >](#)

6 пунктов, которые помогут легко разобраться с regex
proglab.io > p/learn-regex/ ▾

Нашлось 154 млн результатов
17 тыс. показов в месяц
[Дать объявление](#)



ЧТО ТАКОЕ "РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ"?

[Википедия:](#)

Регулярные выражения - формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов.

Это общее название технологии для манипуляций с подстроками.



ЧТО ТАКОЕ "РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ"?

[MDN Web Docs:](#)

Регулярные выражения - это шаблоны используемые для сопоставления последовательностей символов в строках.

А также частное название самих выражений, на которых основана технология.

ЧТО ТАКОЕ "РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ"?

Ну, сопоставление последовательности символов мы выполнять умеем и без этих регулярных выражений.

Например, нам надо произвести валидацию адреса электронной почты: проверить, что адрес электронной почты содержит знак @.

Каким способом можно это сделать?

Можно перебрать каждый символ (самое неправильное решение):

```
1  function validateEmail(emailStr) {
2    for (const itemSymbol of emailStr) {
3      if (itemSymbol === '@') {
4        return true;
5      }
6    }
7    return false;
8  }
9
10 console.log(validateEmail('support@netology.ru'));
11 console.log(validateEmail('supportnetology.ru'));
12
13 // -> true
14 // -> false
```

Можно перебрать каждый символ (самое неправильное решение):

```
1  function validateEmail(emailStr) {  
2      return emailStr.indexOf('@') !== -1;  
3  }  
4  
5  console.log(validateEmail('support@netology.ru'));  
6  console.log(validateEmail('supportnetology.ru'));  
7  
8  // -> true  
9  // -> false
```

Можно же найти по шаблону-регулярному выражению (пока непонятно, лучше ли):

```
1  function validateEmail(emailStr) {  
2      return emailStr.search(/@/) !== -1;  
3  }  
4  
5  console.log(validateEmail('support@netology.ru'));  
6  console.log(validateEmail('supportnetology.ru'));  
7  
8  // -> true  
9  // -> false
```

В реальных задачах редко требуется найти совпадение лишь по одному символу, зачастую требуется, чтобы строка несколькими условиями.

Например, чтобы понять, насколько жизнеспособно каждое из наших решений, давайте введем еще одно правило - в адресе электронной почты после собаки должна присутствовать точка.

Первое выражение примет прямо-таки страшный вид:

```
1  function validateEmail(emailStr) {
2      let foundComercialAt = false;
3      for (const itemSymbol of emailStr) {
4          if (itemSymbol === '@') {
5              foundComercialAt = true;
6          }
7          if ((itemSymbol === '.') && (foundComercialAt)) {
8              return true;
9          }
10     }
11     return false;
12 }

13
14 console.log(validateEmail('support@netology.ru'));
15 console.log(validateEmail('support@netologyru'));
16 console.log(validateEmail('supportnetology.ru'));
```

Второе решение попроще:

```
1 function validateEmail(emailStr) {  
2   const commercialAtPos = emailStr.indexOf('@');  
3   return (commercialAtPos !== -1) &&  
4     (commercialAtPos < emailStr.indexOf('.'));  
5 }  
6  
7 console.log(validateEmail('support@netology.ru'));  
8 console.log(validateEmail('support@netologyru'));  
9 console.log(validateEmail('supportnetology.ru'));
```

Решение с регулярным выражением же немного удивляет:

```
1  function validateEmail(emailStr) {  
2      return emailStr.search(/@\w+\./) !== -1;  
3  }  
4  
5  console.log(validateEmail('support@netology.ru'));  
6  console.log(validateEmail('support@netologyru'));  
7  console.log(validateEmail('supportnetology.ru'));
```

Если же еще ввести требования, что собака не является первым символом адреса, а точка не является последней..

Первый и второй способ станут еще сложнее.

Первый способ (не делайте такого, пожалуйста, в своей работе):

```
1  function validateEmail(emailStr) {  
2    let foundComercialAt = false;  
3    const cuttedEmailStr = emailStr.substring(1, emailStr.length - 1);  
4    for (const itemSymbol of cuttedEmailStr) {  
5      if (itemSymbol === '@') {  
6        foundComercialAt = true;  
7      }  
8      if ((itemSymbol === '.') && (foundComercialAt)) {  
9        return true;  
10     }  
11   }  
12   return false;  
13 }
```

```
console.log(validateEmail('support@netology.ru'));  
console.log(validateEmail('support@netologyru'));  
console.log(validateEmail('@supportnetology.ru'));  
console.log(validateEmail('support@netologyru.'));  
console.log(validateEmail('supportnetology.ru'));  
console.log(validateEmail('supportnetologyru'));  
  
// -> true  
// -> false  
// -> false  
// -> false  
// -> false  
// -> false
```

Второй способ:

```
1 function validateEmail(emailStr) {  
2   const commercialAtPos = emailStr.indexOf("@");  
3   const dotPos = emailStr.indexOf(".");  
4   if (commercialAtPos <= 0) {  
5     return false;  
6   }  
7   if (dotPos < commercialAtPos) {  
8     return false;  
9   }  
10  if (dotPos === emailStr.length - 1) {  
11    return false;  
12  }  
13  return true;  
14 }
```

```
console.log(validateEmail('support@netology.ru'));  
console.log(validateEmail('support@netologyru'));  
console.log(validateEmail('@supportnetology.ru'));  
console.log(validateEmail('support@netologyru.'));  
console.log(validateEmail('supportnetology.ru'));  
console.log(validateEmail('supportnetologyru'));
```

```
// -> true  
// -> false  
// -> false  
// -> false  
// -> false  
// -> false
```

Третий же притерпит всего пару изменений:

```
1  function validateEmail(emailStr) {
2      return emailStr.search(/\w@\w+\.\w/) !== -1;
3  }
4
5  console.log(validateEmail('support@netology.ru'));
6  console.log(validateEmail('support@netologyru'));
7  console.log(validateEmail('@supportnetology.ru'));
8  console.log(validateEmail('support@netologyru. '));
9  console.log(validateEmail('supportnetology.ru'));
10 console.log(validateEmail('supportnetologyru'));
11
12 // -> true
13 // -> false
14 // -> false
15 // -> false
16 // -> false
17 // -> false
```

РЕГУЛЯРНОЕ ВЫРАЖЕНИЕ - ЭТО ШАБЛОН ПОДСТРОКИ.

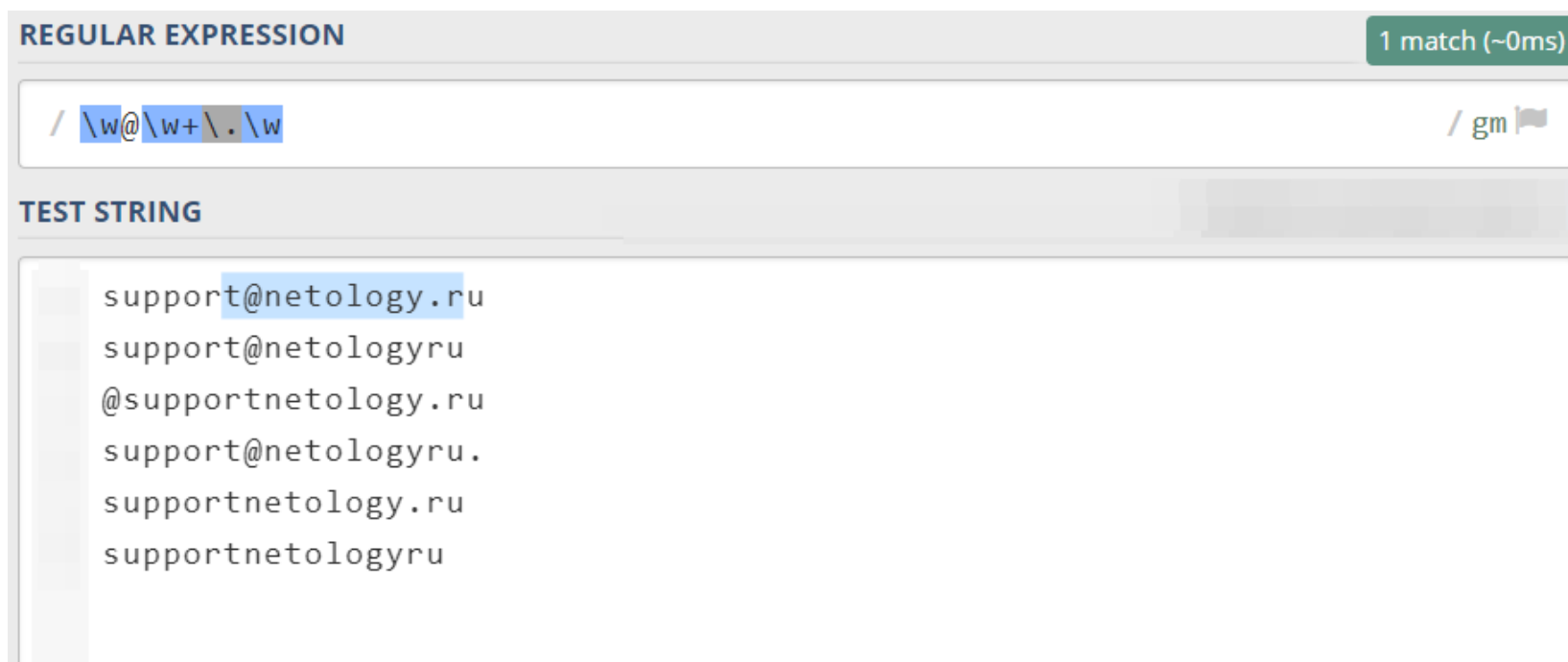
Например, строка `\w@\w+\.\w` читается как:

1. `\w` - любая цифра, буква или знак подчеркивания
2. `@` - символ @
3. `\w+` - любая цифра, буква или знак подчеркивания, один символ или более
4. `\.` - точка

И функция `search()` производит поиск последовательности, подходящей под шаблон в строке `emailStr`

РЕГУЛЯРНОЕ ВЫРАЖЕНИЕ - ЭТО ШАБЛОН ПОДСТРОКИ.

С помощью сайта <https://regex101.com> можно посмотреть совпадение подстроки с шаблоном.



The screenshot shows the regex101.com interface. At the top, the title "REGULAR EXPRESSION" is displayed. To the right of the title, a green badge indicates "1 match (~0ms)". Below the title, the regular expression `/\w@\w+\.\w/` is entered in the input field. To the right of the input field, the flags `/gm` are selected. Below the input field, the title "TEST STRING" is displayed. Underneath, a list of test strings is shown, with the first string, `support@netology.ru`, highlighted in blue, indicating it is the only match for the given regular expression.

REGULAR EXPRESSION

1 match (~0ms)

`/\w@\w+\.\w/` `/gm`

TEST STRING

support@netology.ru
support@netologyru
@supportnetology.ru
support@netologyru.
supportnetology.ru
supportnetologyru

СИНТАКСИС

Синтаксис регулярок (регулярных выражений) достаточно прост.

`/регулярное выражение/` записывается между символами `"/"`

например: `/my_regexp/`

регулярное выражение записывается с помощью специального языка.

Например:

- `\d` - десятичная цифра
- `\D` - любой символ, кроме десятичной цифры

Многие символы в регулярках можно писать явно:

- `a` - символ "a"
- `Я` - буква "Я"



РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

У регулярных выражений достаточно большое количество различных возможностей, все их осветить в рамках данной лекции не удастся, однако изучить их стоит.

Сам язык регулярных выражений во многих языках программирования единый, поэтому освоить его стоит в любом случае.

В конце презентации есть шпаргалка по возможностям регулярок.

В ходе занятия же мы рассмотрим некоторые из них при знакомстве с функциями.

STR.SEARCH(REGEXP)

Этот метод возвращает позицию первого совпадения с шаблоном регулярного выражения.

Если совпадения нет, то результатом выполнения будет **-1**

Синтаксис:

```
str.search(RegExp);
```

Один пример использования рассмотрен выше.

STR.SEARCH(REGEXP)

```
1  function validateEmail(emailStr) {  
2      return emailStr.search(/\w@\w+\.\w/) !== -1;  
3  }  
4  
5  console.log(validateEmail('support@netology.ru'));  
6  console.log(validateEmail('support@netologyru'));  
7  console.log(validateEmail('@supportnetology.ru'));  
8  console.log(validateEmail('support@netologyru.'));  
9  console.log(validateEmail('supportnetology.ru'));  
10 console.log(validateEmail('supportnetologyru'));
```

В этом примере производится поиск совпадения с шаблоном. Если поиск удачен, то возвращается число большее или равное нулю, на что и производится проверка в условии.

STR.MATCH(REGEXP)

Возвращает результат совпадения с шаблоном.

Попробуем записать предыдущую проверку через `match()`. Для наглядности пока просто выведем результат выполнения `match()`

```
1  function validateEmail(emailStr) {  
2      return emailStr.match(/\w@\w+\.\w/);  
3  }  
4  
5  console.log(validateEmail('support@netology.ru'));  
6  console.log(validateEmail('support@netologyru'));  
7  console.log(validateEmail('@supportnetology.ru'));  
8  console.log(validateEmail('support@netologyru.'));  
9  console.log(validateEmail('supportnetology.ru'));  
10 console.log(validateEmail('supportnetologyru'));
```

STR.MATCH(REGEXP)

Если `search()` оповещает только о факте совпадения, то `match()` нам показывает и сам результат сверки с шаблоном.

```
>
function emailValidate(emailStr){
    return emailStr.match(/\w@\w+\.\w/);
}

console.log(emailValidate('support@netology.ru'));
console.log(emailValidate('support@netologyru'));
console.log(emailValidate('@supportnetology.ru'));
console.log(emailValidate('support@netologyru. '));
console.log(emailValidate('supportnetology.ru'));
console.log(emailValidate('supportnetologyru'));
```

VM672:6

```
▼ ["t@netology.r", index: 6, input: "support@netology.ru", groups: undefined] ⓘ
  0: "t@netology.r"
  groups: undefined
  index: 6
  input: "support@netology.ru"
  length: 1
  ▶ __proto__: Array(0)
```

null VM672:7

null VM672:8

null VM672:9

null VM672:10

null VM672:11

STR.MATCH(REGEXP)

Проверка приобретает такой вид:

```
1  function validateEmail(emailStr) {  
2      return emailStr.match(/\w@\w+\.\w/) !== null;  
3  }  
4  
5  console.log(validateEmail('support@netology.ru'));  
6  console.log(validateEmail('support@netologyru'));  
7  console.log(validateEmail('@supportnetology.ru'));  
8  console.log(validateEmail('support@netologyru.'));  
9  console.log(validateEmail('supportnetology.ru'));  
10 console.log(validateEmail('supportnetologyru'));
```

STR.MATCH(REGEXP)

Давайте изменим нашу регулярку:

```
/^\w+@\w+\.\w+$/
```

^ обозначает начало строки (не внутри скобок)

\$ обозначает конец строки

Это позволит нам валидировать строку полностью.

STR.MATCH(REGEXP)

```
1  function validateEmail(emailStr) {  
2      return emailStr.match(/^\\w+@\\w+\\.\\w+$/ ) !== null;  
3  }  
4  
5  console.log(validateEmail('support@netology.ru'));  
6  console.log(validateEmail('suppart@netologyru'));  
7  console.log(validateEmail('$support@netologyru'));  
8  console.log(validateEmail('support@netologyru'));  
9  console.log(validateEmail('@supportnetology.ru'));  
10 console.log(validateEmail('support@netologyru. '));  
11 console.log(validateEmail('supportnetology.ru'));  
12 console.log(validateEmail('supportnetologyru'));
```


STR.SPLIT(REGEXP|STR, LIMIT)

Функция `split()` позволяет разбить строку по какому-либо разделителю.

Как разделитель можно использовать как строку, так и регулярку.

`limit` - необходимое количество полученных элементов массива, по умолчанию - неограничено.

STR.SPLIT(REGEXP|STR, LIMIT)

Разобьем предложение на слова:

```
1 function separatePhrase(phrase) {  
2   return phrase.split(/^[^а-яёа-z@\.]*/i);  
3 }  
4 console.log(separatePhrase('Support@netology.ru \\  
5 - адрес технической поддержки Нетологии.'));
```

^ внутри скобок используется для отрицания

[] группа возможных символов

а-я указывает на диапазон символов (от "а" до "я")

Флаг **i** указывает на игнорирование регистра.



STR.REPLACE(REGEXP, STR|FUNC)

Чудо, а не функция.

Позволяет произвести определенные операции с определенными участками текста.



STR.REPLACE(REGEXP, STR|FUNC)

Давайте напишем функцию перевода на "кирпичный язык".

Кто помнит, как перевести на "кирпичный"? Можно просто пример.

STR.REPLACE(REGEXP, STR|FUNC)

В круглых скобках указывается группа символов.

Для получения выбранной группы используем `$1`

```
1 function transferToBrick(phrase) {  
2   return phrase.toUpperCase().replace(/([АЯЭЕОЁУЮЫИ])/, '$1K$1');  
3 }  
4 console.log(transferToBrick('Привет, мир!'));
```

STR.REPLACE(REGEXP, STR|FUNC)

Флаг **g** указывает на то, что поиск будет производиться по всей фразе.

Без этого флага будет изменен только первый символ.

```
1 function transferToBrick(phrase) {  
2   return phrase.toUpperCase().replace(/([АЯЭЕОЁУЮЫИ])/g, '$1K$1');  
3 }  
4 console.log(transferToBrick('Привет, мир!'));
```

REGEXP.TEST(STR)

Функция, схожая с `str.search(regex) !== -1`. Проверяет, есть ли хоть одно совпадение.

```
1  function validateEmail(emailStr) {  
2      return /\w@\w+\.\w/.test(emailStr);  
3  }  
4  
5  console.log(validateEmail('support@netology.ru'));  
6  console.log(validateEmail('support@netologyru'));  
7  console.log(validateEmail('@supportnetology.ru'));  
8  console.log(validateEmail('support@netologyru.'));  
9  console.log(validateEmail('supportnetology.ru'));  
10 console.log(validateEmail('supportnetologyru'));
```

REGEXP.EXEC(STR)

Функция, похожая на `str.match(RegExp)`.

Возвращает совпадение с шаблоном

```
1  function validateEmail(emailStr) {  
2      return /^\\w+@\\w+\\.\\w+$/ .exec(emailStr) !== null;  
3  }  
4  
5  console.log(validateEmail('support@netology.ru'));  
6  console.log(validateEmail('suppart@netologyru'));  
7  console.log(validateEmail('$support@netologyru'));  
8  console.log(validateEmail('support@netologyru'));  
9  console.log(validateEmail('@supportnetology.ru'));  
10 console.log(validateEmail('support@netologyru.'));  
11 console.log(validateEmail('supportnetology.ru'));  
12 console.log(validateEmail('supportnetologyru'));
```


REGEXP.EXEC(STR)

Кстати, если нам потребуется извлечь адрес электронной почты из предложения:

Возвращает совпадение с шаблоном

```
1 function findEmail(emailStr) {  
2     return /\w+@\w+\.\w+/.exec(emailStr);  
3 }  
4  
5 console.log(findEmail('Support@netology.ru \  
6 - адрес технической поддержки Нетологии.')[0]);  
7  
8 // -> Support@netology.ru
```

Аналогичный результат принесёт и функция `str.match(RegExp)`

REGEXP.TOSTRING()

Возвращает строковое представление регулярного выражения.

```
1  const myReg = /^\\w+@\\w+\\.\\w+$/g;  
2  console.log(typeof(myReg));  
3  console.log(typeof(myReg.toString()));  
4  
5  // -> object  
6  // -> string
```

ВОЗМОЖНОСТИ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ В СТАНДАРТЕ ES2018

Эти возможности установлены стандартом ES2018, но ещё (на момент написания лекции) реализованы не во всех браузерах.

- именованные группы;
- `±lookbehind`, `±lookahead`;
- флаг `s`;
- паттерны для работы с Unicode;

Эти возможности установлены стандартом ES2018, но ещё (на момент написания лекции) реализованы не во всех браузерах.

ИМЕНОВАННЫЕ ГРУППЫ

В "js будущего" появилась возможность давать группам наименования.

```
1  function findEmail(emailStr) {  
2      return /(?!<emailGroup>\w+@\w+\.\w+)/.exec(emailStr);  
3  }  
4  
5  const textStr = 'Support@netology.ru \\  
6  - адрес технической поддержки Нетологии.';  
7  const emailStr = findEmail(textStr);  
8  console.log(emailStr.groups.emailGroup);  
9  
10 // -> Support@netology.ru
```

LOOKBEHIND, LOOKAHEAD

"Взгляд назад" и "Взгляд вперёд":

- `x(?=y)` - ищет соответствие паттерну `x`, когда он идёт перед `y` (положительная опережающая проверка);
- `x(?!y)` - ищет соответствие паттерну `x`, когда он идёт не перед `y` (негативная опережающая проверка);
- `(?<=y)x` - ищет соответствие паттерну `x`, когда он идёт после `y` (положительная ретроспективная проверка);
- `(?<!y)x` - ищет соответствие паттерну `x`, когда он идёт не после `y` (негативная ретроспективная проверка);

LOOKBEHIND, LOOKAHEAD

```
1 function findEmail(emailStr) {  
2   return emailStr.match(/\w+(?=@)/g);  
3 }  
4  
5 textStr = 'admin@netology, email: support@netology.ru';  
6 console.log(findEmail(textStr));  
7  
8 // -> ["admin", "support"]
```

ФЛАГ `s` (DOTALL)

Хотя и считается, что символ точки соответствует любому одиночному символу, он не соответствует некоторым символам, например, символу перевода строки `\n`.

```
1 function matchPhrase(phraseStr) {  
2   return /Нетология.онлайн-школа/.exec(phraseStr);  
3 }  
4  
5 const textStr = 'Нетология\nонлайн-школа';  
6 console.log(matchPhrase(textStr));  
7  
8 // -> null
```

ФЛАГ `s` (DOTALL)

Флаг `s` позволяет видеть как точку абсолютно любой символ:

```
1 function matchPhrase(phraseStr) {  
2   return /Нетология.онлайн-школа/s.exec(phraseStr);  
3 }  
4  
5 const textStr = 'Нетология\нонлайн-школа';  
6 console.log(matchPhrase(textStr));  
7  
8 // -> ["Нетология.онлайн-школа", index: 0,  
9 // input: "Нетология\нонлайн-школа", groups: undefined]
```


ПАТТЕРНЫ ДЛЯ РАБОТЫ С UNICODE-СИМВОЛАМИ

Были расширены возможности для работы с Unicode:

```
1 console.log(/\p{Emoji}/u.test('😄ΩU'));
2 console.log(/\p{Script=Greek}/u.exec('😄ΩU'));
3
4 // -> true
5 // -> ["Ω", index: 0, input: "😄ΩU", groups: undefined]
```

РЕГУЛЯРНОЕ ВЫРАЖЕНИЕ - ЗЛО, ЕСЛИ...

Регулярные выражения - удобный, полезный, высокопроизводительный и простой инструмент.

Однако, важно помнить, что:

- как и любой другой инструмент, он НЕ универсален;
- регулярное выражение сложно читаемо.

Достаточно часто разработчики могут создать "дьявольское" регулярное выражение.

Так же, среди разработчиков бытует мнение, что "регулярки пишутся в одну сторону" - то есть их пишут, но не читают, регулярное выражение проще написать с нуля, чем разобраться в готовом.

Поэтому,

РЕГУЛЯРНОЕ ВЫРАЖЕНИЕ - ЗЛО, ЕСЛИ:

1. Есть более подходящие инструменты.

Например, для проверки кода есть большое количество готовых синтаксических анализаторов. Неправильно проверять "правильно ли сформирован JSON" собственной регуляркой.

Например, если требуется только определить наличие единственного символа в строке, с этим справится и функция `str.indexOf()`.

Поэтому,

РЕГУЛЯРНОЕ ВЫРАЖЕНИЕ - ЗЛО, ЕСЛИ:

1. Есть более подходящие инструменты.

Например, Вам требуется найти наличие двух подстрок в тексте. Может быть, код для поиска двух подстрок будет эффективнее, чем одна регулярка?

Например, не пытайтесь проверить текст на цензурность с помощью одной регулярки. Эта проблема эффективнее решается несколькими различными инструментами, где регулярным выражениям отведена вспомогательная роль.

РЕГУЛЯРНОЕ ВЫРАЖЕНИЕ - ЗЛО, ЕСЛИ:

2. Вы пытаетесь решить одной регуляркой задачу, которую НУЖНО решать несколькими регулярными выражениями

Например, Вам требуется проверить серию и номер документа. Если у вас допускается два документа (например, паспорт и свидетельство о рождении), не стоит пытаться объединить два шаблона для проверки в один. Даже если Вы получите выигрыш в производительности (что маловероятно), Вы существенно потеряете в читаемости кода.



РЕГУЛЯРНОЕ ВЫРАЖЕНИЕ - ЗЛО, ЕСЛИ:

3. Регулярное выражение нечитаемо.

Например, с вероятностью 99,9% другой разработчик не станет разбираться в смысле следующей регулярки:

```

(( [0-9a-fA-F]{1,4}: ){7,7} [0-9a-fA-F]{1,4}
| ([0-9a-fA-F]{1,4}: ){1,7}: | ([0-9a-fA-F]{1,4}: )
{1,6}: [0-9a-fA-F]{1,4} | ([0-9a-fA-F]{1,4}: ){1,5}
(: [0-9a-fA-F]{1,4}) {1,2} | ([0-9a-fA-F]{1,4}: ){1,4}
(: [0-9a-fA-F]{1,4}) {1,3} | ([0-9a-fA-F]{1,4}: ){1,3}
(: [0-9a-fA-F]{1,4}) {1,4} | ([0-9a-fA-F]{1,4}: ){1,2}
(: [0-9a-fA-F]{1,4}) {1,5} | [0-9a-fA-F]{1,4}:
(( (: [0-9a-fA-F]{1,4}) {1,6} ) | (: (: [0-9a-fA-F]{1,4})
{1,7} | : ) | fe80: (: [0-9a-fA-F]{0,4}) {0,4}
% [0-9a-zA-Z]{1,} | :: (ffff (: 0{1,4}) {0,1}: ) {0,1}
(( 25[0-5] | ( 2[0-4] | 1{0,1} [0-9] ) ) {0,1} [0-9] ) \. ) {3,3}
( 25[0-5] | ( 2[0-4] | 1{0,1} [0-9] ) ) {0,1} [0-9] ) |
([0-9a-fA-F]{1,4}: ){1,4}: (( 25[0-5] | ( 2[0-4] | 1{0,1}
[0-9] ) ) {0,1} [0-9] ) \. ) {3,3} ( 25[0-5] | ( 2[0-4] | 1{0,1}
[0-9] ) ) {0,1} [0-9] ))

```



РЕГУЛЯРНОЕ ВЫРАЖЕНИЕ - ЗЛО, ЕСЛИ:

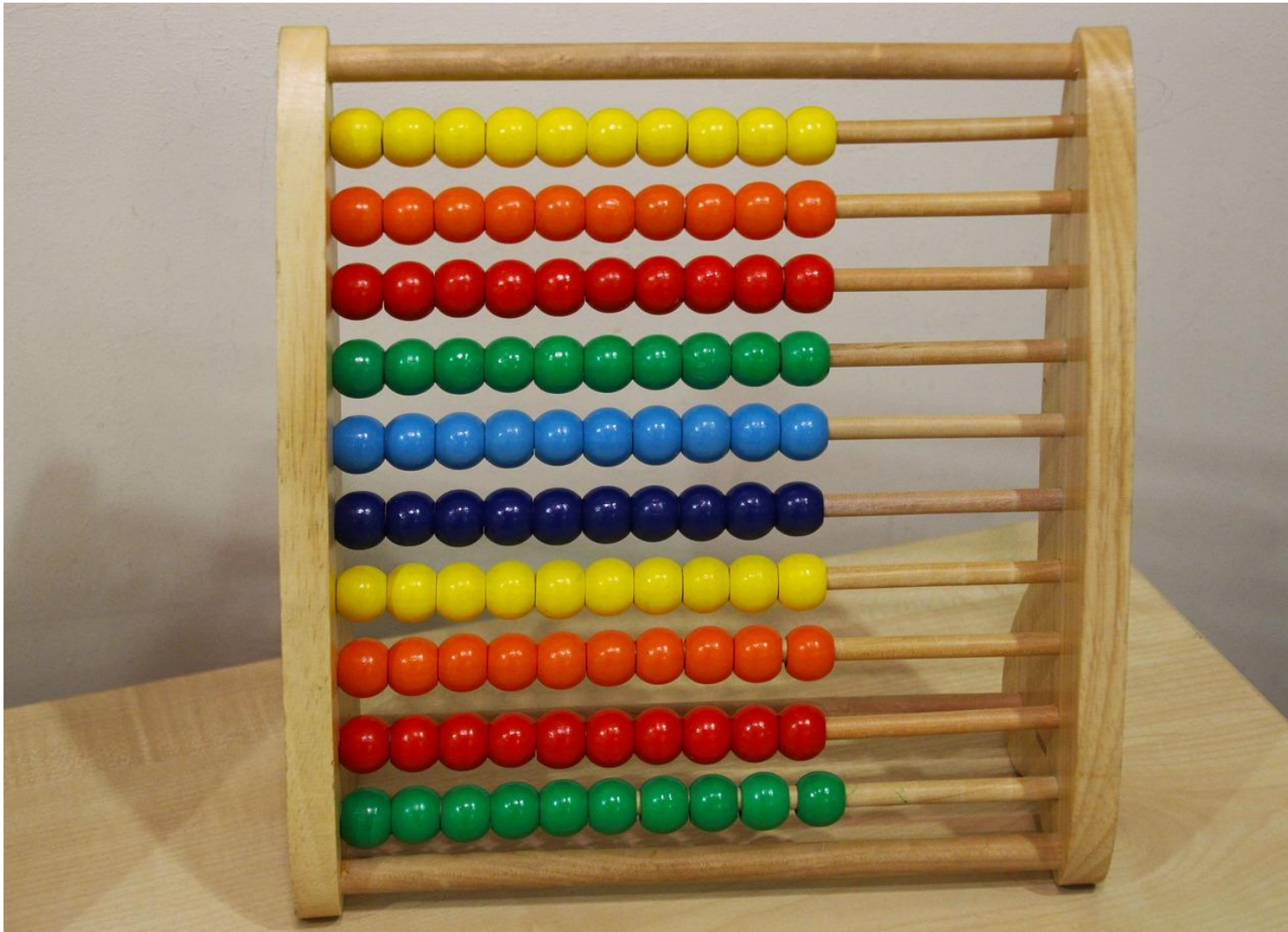
4. Вы не умеете писать регулярные выражения или Ваш код "грязный".

Если регулярное выражение не работает - оно вредит.

Если регулярное выражение слишком требовательное к ресурсам - оно вредит.

Если в мешанине кода всплывает регулярное выражение - оно вредит.

MATH



Объект `Math` содержит набор математических функций и констант.

СВОЙСТВА (МАТЕМАТИЧЕСКИЕ КОНСТАНТЫ):

`Math.E` - число Эйлера, так же известное, как математическая константа, обозначаемая символом e .

`Math.PI` - отношение длины окружности её диаметру, так же известно как "число пи".

`Math.LN2` - натуральный логарифм из 2.

`Math.LN10` - натуральный логарифм из 10.

`Math.LOG2E` - двоичный логарифм из числа Эйлера.

`Math.LOG10E` - десятичный логарифм из числа Эйлера.

`Math.SQRT1_2` - квадратный корень из $1/2$.

`Math.SQRT2` - квадратный корень из 2.

ОКРУГЛЕНИЕ:

`Math.ceil(x)` - возвращает наименьшее целое число, большее, либо равное указанному числу ("округление вверх").

`Math.floor(x)` - возвращает наибольшее целое число, меньшее, либо равное указанному числу ("округление вниз").

`Math.round(x)` - возвращает значение числа, округлённое до ближайшего целого.

`Math.trunc(x)` - возвращает целую часть числа, убирая дробные цифры.

ТРИГОНОМЕТРИЧЕСКИЕ ФУНКЦИИ:

`Math.acos(x)` - возвращает арккосинус числа.

`Math.asin(x)` - возвращает арксинус числа.

`Math.atan(x)` - возвращает арктангенс числа.

`Math.cos(x)` - возвращает косинус числа.

`Math.sin(x)` - возвращает синус числа.

`Math.tan(x)` - возвращает тангенс числа.

Внимание! Тригонометрические функции принимают в параметрах или возвращают углы в радианах.

ТРИГОНОМЕТРИЧЕСКИЕ ФУНКЦИИ:

`Math.abs(x)` - возвращает абсолютное значение (модуль) числа.

`Math.sqrt(x)` - возвращает положительный квадратный корень числа.

`Math.cbrt(x)` - возвращает кубический корень числа.

`Math.max([x[, y[, ...]]])` - возвращает наибольшее число из своих аргументов.

`Math.min([x[, y[, ...]]])` - возвращает наименьшее число из своих аргументов.

`Math.random()` - возвращает псевдослучайное число в диапазоне от 0 до 1.

`Math.log(x)` - возвращает натуральный логарифм числа.



MATH

Это далеко не полный перечень.

Современный объект `Math` дает полноценный математический инструментарий.

Чтобы использовать этот инструмент, требуется обратиться к объекту

Math

```
1  const randomDigit = Math.random();
2  console.log(randomDigit);
3
4  const xGrad = Math.round(randomDigit * 180);
5  console.log(xGrad);
6
7  const xRad = xGrad * Math.PI / 180;
8  console.log(xRad);
9
10 const cosX = Math.cos(xRad);
11 const sinX = Math.sin(xRad);
12
13 console.log(cosX);
14 console.log(sinX);
15 console.log(cosX**2 + sinX**2);
```


MATH

Как найти длину гипотенузы при известных катетах через JS?

Напомню, квадрат гипотенузы равен сумме квадратов катетов

```
1  const cathetusFirst = 3;  
2  const cathetusSecond = 4;  
3  const hypotenuse = ...;  
4  
5  console.log(hypotenuse);
```

```
1  const cathetusFirst = 3;  
2  const cathetusSecond = 4;  
3  const hypotenuse = Math.hypot(cathetusFirst, cathetusSecond);  
4  
5  console.log(hypotenuse);
```

ARRAYBUFFER



УПРАВЛЕНИЕ ПАМЯТЬЮ

Автоматическое управление памятью упрощает разработку приложения, однако снижает производительность программ.

Например, когда вы создаёте переменную, js-движок определяет, какого типа будет эта переменная, какой объем памяти ей потребуется.

Достаточно часто это ведет к резервированию большего объема памяти, чем на самом деле нужно для хранения переменной. При этом требуемый размер может быть в 2-8 раз меньше, чем резервируемый. Это приводит к неэффективному использованию памяти.



УПРАВЛЕНИЕ ПАМЯТЬЮ

Во многих случаях, автоматическое управление памятью не вызывает проблем. Большинство JS-приложений не настолько требовательны к производительности, чтоб им было необходимо ручное управление памятью. Однако, ручное управление памятью негативно влияет на производительность труда программиста.

Случаи, в которых требуется ручное управление памятью - это случаи требовательных к оперативной памяти js-приложений.



ARRAYBUFFER

Как одно из решений для ручного контроля управления памятью можно рассмотреть `ArrayBuffer`.

NEW ARRAYBUFFER

ArrayBuffer представляет собой ссылку на поток "сырых" данных.

Например, если создать `ArrayBuffer` длины 4:

```
1 | const buffer = new ArrayBuffer(4);
```

Мы получим просто данные следующего вида (в байтах): `0000`

ПРЕДСТАВЛЕНИЕ ARRAYBUFFER

Эти данные можно представить в разных вариантах: разделив по байту, 2 байта, или четыре.

Чтобы представить `ArrayBuffer` в каком-либо виде, требуется создать его представление:

```
1  const buffer = new ArrayBuffer(4);
2  const buffer8BitView = new Int8Array(buffer);
3
4  console.log(buffer8BitView);
5
6  // -> Int8Array(4)[0, 0, 0, 0];
```


ПРЕДСТАВЛЕНИЕ ARRAYBUFFER

Этому же буферу можно задать и другое представление:

```
1  const buffer = new ArrayBuffer(4);
2  const buffer8BitView = new Int8Array(buffer);
3  const buffer16BitView = new Int16Array(buffer);
4
5  console.log(buffer8BitView);
6  console.log(buffer16BitView);
7
8  // -> Int8Array(4)[0, 0, 0, 0];
9  // -> Int16Array(2)[0, 0];
```

ПРЕДСТАВЛЕНИЕ ARRAYBUFFER

После создания представления можно в `ArrayBuffer` внести данные или изъять из него:

```
1  const buffer = new ArrayBuffer(4);
2  const buffer8BitView = new Int8Array(buffer);
3  const buffer16BitView = new Int16Array(buffer);
4
5  buffer16BitView[0] = 1000;
6
7  console.log(buffer16BitView[0] + 1);
8
9  // -> 1001
```

ПРЕДСТАВЛЕНИЕ ARRAYBUFFER

Обратите внимание, что доступ к одному представлению не прекращается при использовании другого:

```
1  const buffer = new ArrayBuffer(4);
2  const buffer8BitView = new Int8Array(buffer);
3  const buffer16BitView = new Int16Array(buffer);
4
5  buffer16BitView[0] = 1000;
6
7  console.log(buffer8BitView);
8  console.log(buffer16BitView);
9
10 // -> Int8Array(4) [-24, 3, 0, 0]
11 // -> Int16Array(2) [1000, 0]
```

ПРЕДСТАВЛЕНИЕ ARRAYBUFFER

Доступны следующие представления:

`Int8Array()` - восьмибитное число со знаком

`Uint8Array()` - беззнаковое восьмибитное число

`Uint8ClampedArray()` - беззнаковое восьмибитное число ("зажатое")

`Int16Array()` - шестнадцитибитное число со знаком

`Uint16Array()` - беззнаковое шестнадцитибитное число

`Int32Array()` - 32-битное число со знаком

`Uint32Array()` - беззнаковое 32-битное число

`Float32Array()` - 32-битное вещественное число

`Float64Array()` - 64-битное вещественное число

ПРЕДСТАВЛЕНИЕ ARRAYBUFFER

Чтоб понять разницу между `Uint8Array` и `Uint8ClampedArray`, давайте проведем следующий опыт:

```
1  const buffer = new ArrayBuffer(2);
2  const notClampedBufferView = new Uint8Array(buffer);
3  const clampedBufferView = new Uint8ClampedArray(buffer);
4
5  console.log('Step 1');
6  notClampedBufferView[0] = 100;
7  clampedBufferView[1] = 100;
8  console.log(notClampedBufferView[0]);
9  console.log(clampedBufferView[1]);
10 console.log('-----');
```

```
1 console.log('Step 2');
2 notClampedBufferView[0] += 100;
3 clampedBufferView[1] += 100;
4 console.log(notClampedBufferView[0]);
5 console.log(clampedBufferView[1]);
6 console.log('-----');
7
8 console.log('Step 3');
9 notClampedBufferView[0] += 100;
10 clampedBufferView[1] += 100;
11 console.log(notClampedBufferView[0]);
12 console.log(clampedBufferView[1]);
13 console.log('-----');
```



ПРЕДСТАВЛЕНИЕ ARRAYBUFFER

Что произошло?

Почему такая разница?

ПРЕДСТАВЛЕНИЕ ARRAYBUFFER

Не стоит забывать, что в ArrayBuffer можно хранить и двоичное представление других (нечисловых) данных:

```
1  const helloStr = 'Hello, world!';
2
3  const buffer = new ArrayBuffer(helloStr.length);
4  const bufferView = new Uint8Array(buffer);
5
6
7  for (let i = 0; i < bufferView.length; i += 1) {
8      bufferView[i] = helloStr.charCodeAt(i);
9  }
10
11 for (let i = 0; i < bufferView.length; i += 1) {
12     console.log(String.fromCharCode(bufferView[i]));
13 }
```




ИТАК, ПОДВЕДЁМ ИТОГИ

На этой лекции были рассмотрены следующие инструменты:

- Регулярные выражения
- Объект `Math`
- Объект `ArrayBuffer`

ИНТЕРЕСНОЕ ЧТИВО

Регулярные выражения:

- [Шпаргалка по регулярным выражениям](#)
- [regex101.com - проверить свою регулярку](#)
- [MDN - сводка по RegExp](#)
- [MDN - Руководство по JavaScript: Регулярные выражения](#)
- [Пример, так делать НЕЛЬЗЯ](#)
- [Обзор новшеств ES2016-ES2018](#)

Math:

- [MDN - Math](#)

ArrayBuffer:

- [MDN - Типизованные массивы JavaScript](#)
- [Habr - ArrayBuffer и SharedArrayBuffer](#)



Спасибо за внимание!!! Жду ваших вопросов 😊

ЛЕКТОР

