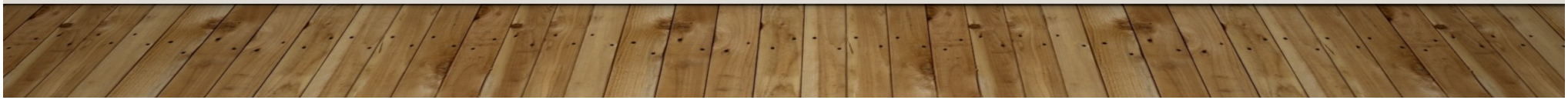
ЛЕКЦИЯ 18

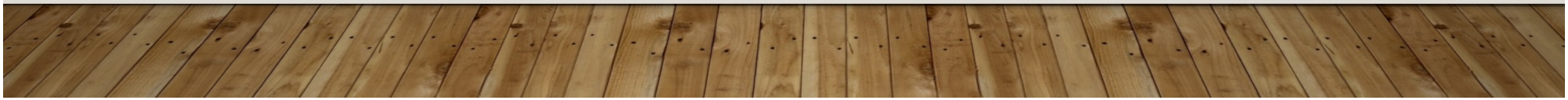
РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ В RUBY



ПЛАН ЗАНЯТИЯ

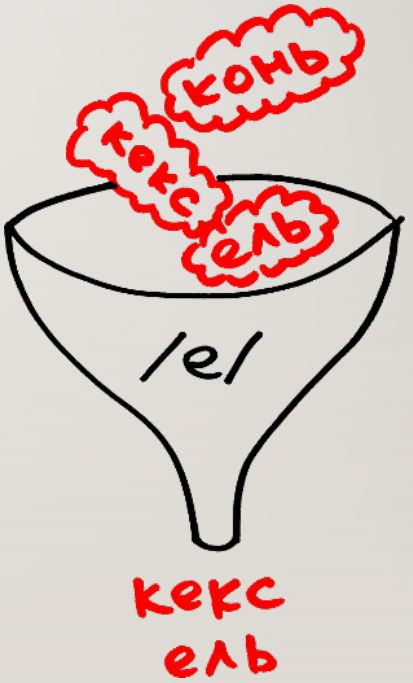
1. Что такое регулярные выражения
2. Разбираем примеры
3. Создаем игру «Придумай слова по шаблону»
4. В каких ситуациях уместно использование регулярных выражений, а в каких лучше обойтись стандартными средствами

- Мы познакомимся и немного поработаем с этим мощным инструментом, разобрав несколько примеров. А потом напишем увлекательную игру, применив полученные знания на практике.



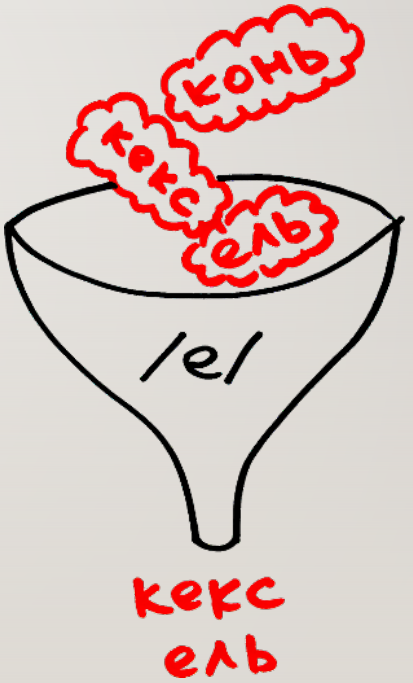
ЧТО ТАКОЕ РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

- Регулярные выражения нужны для работы со строками.
- Их используют, когда надо найти в строке совпадение с каким-то шаблоном, и сделать что-то, если это совпадение найдено, например, заменить часть строки другими символами.
- Можно представить регулярные выражения в виде воронки:



ЧТО ТАКОЕ РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

- Данная воронка пропускает только те слова, которые содержат в себе букву e.
- Регулярное выражение, записанное в слешах //, как бы говорит нам, если найдешь в строке, к которой меня применили, букву, которая у меня между слешей (e), то пропускай его, иначе - нет.
- Для работы с регулярными выражениями, стандартная библиотека Ruby имеет в своем арсенале класс Regexp, а также методы определенные других классах.



ПРИМЕРЫ

- Перед нами уже стояла задача, когда нам необходимо было сравнить строку на наличие определенных символов, но иногда стандартных методов бывает недостаточно, и на помощь приходят регулярные выражения. Для начала давайте рассмотрим несколько простых примеров.
- Начнем с поиска строк уже известным нам способом.
- Здесь мы проверяем строку на наличие слова Маша, результат, соответственно, возвращает нам true:

```
'Маша и Гоша'.include?('Маша')
=> true
```

ПРИМЕРЫ

- А в данном случае слово Саша в нашей строке отсутствует — получаем результат false:

```
'Маша и Гоша'.include?('Саша')
=> false
```

ПРИМЕРЫ

- Теперь на этом же примере воспользуемся регулярным выражением, а именно задействуем метод match, аргументом которого является искомое значение объятое слешами, т.е. /Гоша/. Метод возвращает нам найденное значение в виде объекта класса MatchData, а конкретнее имя 'Гоша' в закодированном виде:

```
'Маша и Гоша'.match(/Гоша/)
=> #<MatchData "\x83\xAE\xE8\xA0">
```

ПРИМЕРЫ

- В случае же использования слова, которое строка не содержит мы получим nil:

```
'Маша и Гоша'.match(/Каша/)
=> nil
```

- Также мы можем воспользоваться более лаконичным методом =~, который проделывает аналогичную работу:

```
'Маша и Гоша' =~ /Каша/
=> nil
```

ПРИМЕРЫ

- Как говорится от перемены мест слагаемых сумма не меняется, так и здесь, мы можем поменять строку с регулярным выражением местами, метод также будет работать:

```
/Каша/ =~ 'Маша и Гоша'
=> nil
```

- Подставляем корректное слово:

```
'Маша и Гоша' =~ /Гоша/
=> 7
```

ПРИМЕРЫ

- И видим, что метод вернул нам 7. Данное число означает позицию (номер символа) в строке, где впервые было встречено совпадение (искомое условие). Давайте проверим, верен ли результат:

```
'Маша и Гоша'.index('Гоша')
=> 7
```

- Как мы видим, результаты совпадают. Отлично, двигаемся дальше.

ПРИМЕРЫ

- Мы можем задавать более сложные конструкции, например нам необходимо найти все числа, или же пропустить только те данные, которые соответствуют регулярному выражению (зачастую Вам придется использовать данный подход при проверке email).
- Рассмотрим следующий пример:

```
'cat' =~ /c.t/
=> 0
```

ПРИМЕРЫ

- Точка в регулярном выражении означает, что на ее месте возможен любой **единственный** символ, будь то **ОДНО** число, **ОДНА** буква или иной символ:

```
'cute' =~ /c.t/
=> 0
```

```
'scute' =~ /c.t/
=> 1
```

```
'caat' =~ /c.t/
=> nil
```

- В последнем примере из-за использования нескольких символов на месте точки вместо одного — мы получили nil.

ПРИМЕРЫ

- А теперь давайте в нашем регулярном выражении применим якорь (*anchor*) ^ — означающий начало строки. При использовании данного якоря, мы явно указываем, что символ c должен идти первым. В случае же иного символа на первом месте после якоря (начала строки) метод вернет нам nil:

```
'cat' =~ /^c.t/
=> 0
```

```
'1cat' =~ /^c.t/
=> nil
```

ПРИМЕРЫ

- А сейчас введем ограничение на последнюю букву, то есть, буква `t` должна быть последней в строке, иначе строка не пройдет нашу 'воронку'. Для этого воспользуемся якорем `$` — обозначающим конец строки:

```
'cat' =~ /^c.t$/
=> 0
```

```
'cute' =~ /^c.t$/
=> nil
```

- В последнем примере буква `t` не является последним символом, отсюда и результат `nil`.

ПРИМЕРЫ

- Пришло время познакомиться с квантификаторами (***quantifier***) от английского слова *quantify* — что означает *определять количество* . Возьмем уже знакомый нам пример, и вместо точки подставим символ звездочки * (*астериск*, *asterisk*). Воспользовавшись данным квантификатором мы можем на место астериска вписать любое количество символов от нуля до бесконечности, и строка пройдет через нашу 'воронку':

```
'cattttttttttttttt' =~ /^c.t*$/
=> 0
```

```
'ca' =~ /^c.t*$/
=> 0
```

ПРИМЕРЫ

- Также существует *квантификатор* +. Он проверяет количество символов от одного до бесконечности:

```
'cat' =~ /^c.+t$/
=> 0
```

```
'ct' =~ /^c.+t$/
=> nil
```

ПРИМЕРЫ

- Рассмотрим ситуацию, когда нам необходимо удостовериться, что строка содержит в себе только числа, для этого воспользуемся следующим регулярным выражением, применив *классы символов* (*character classes*), то есть все то, что находится в квадратных скобках:

```
'01234567890' =~ /^[0-9]*$/
=> 0
```

ПРИМЕРЫ

- А теперь мы укажем определенные числа:

```
'234234234' =~ /^[234]*$/
=> 0
```

- Последовательность чисел не имеет значения:

```
'43233224244' =~ /^[234]*$/
=> 0
```

ПРИМЕРЫ

- Если среди этих чисел окажется что-нибудь другое, например 1 или 5, то результатом будет nil:

```
'2341' =~ /^[234]*$/
=> nil
```

```
'234a' =~ /^[234]*$/
=> nil
```

ПРИМЕРЫ

- Отлично, мы молодцы! Давайте напишем что то более интересное и практичное, а заодно закрепим результат. Провалидируем email, то есть создадим упрощенное регулярное выражение, которое проверяет корректность веденного email:

```
/^[a-z0-9]+@[a-z0-9]+\.[a-z]+/ =~ 'google@gmail.com'
=> 0
```

```
/^[a-z0-9]+@[a-z0-9]+\.[a-z]+/ =~ 'google@gmail.%%'
=> nil
```

ПРИМЕРЫ

- Разберем пример подробнее:
- // — как мы помним, между символами слеш мы помещаем наше регулярное выражение
- ^ — данный якорь означает начало строки
- [a-z0-9] — класс символов, здесь мы указали, что допускается любой числовой символ, а также любая строчная (маленькая, не заглавная) буква английского алфавита. Соответственно, если бы мы хотели пропускать заглавные буквы, в таком случае мы бы написали [A-Z]. Либо, чтобы было еще красивее и компактнее, в конце нашего регулярного выражения мы можем добавить *модификатор* (*modifier*) i, который закрывает глаза на case букв (пропускает как строчные, так и заглавные). Выражение примет вид: /^[a-z0-9]+@[a-z0-9]+\.[a-z]+/i

ПРИМЕРЫ

- Разберем пример подробнее:
- + — квантификатор, пропускающий результат от одного до бесконечности, то есть в нашем случае [a-z0-9]+ — допускается любое количество букв английского алфавита и цифр от 1 до бесконечности
- @ — проверяем, что строка имеет данный символ в строго указанном месте, и этот символ должен быть не больше, и не меньше одного
- \. — мы помним, что . в регулярном выражении означает, что на месте данной точки применим любой символ в единственном числе, но мы то хотим указать конкретный символ точки, который прописывается в email, для этого нам необходимо экранировать нашу точку обратным слешем

ПРИМЕРЫ

- Разберем пример подробнее:
- [a-z]+ — и последнее, проверяем на корректность написания доменного имени верхнего уровня (то есть, например ru или com)

ПРИМЕРЫ

- Мы можем написать наше регулярное выражение немного иначе, воспользовавшись следующим синтаксисом:

```
`/^[\w\d]+@[\w\d]+\.[\w]+/ =~ 'google2012@gmail.com'`
=> 0
```

- где \w — аналогично выражению [a-zA-Z], то есть любая буква английского алфавита, независимая от case, но помимо этого также допускается наличие знака нижнего подчеркивания _, что в общем то нам и требуется для проверки email, поскольку данный символ в адресе электронной почты допустим
- \d — аналогично выражению [0-9], то есть проверяет строку на наличие любых чисел

ПРИМЕРЫ

- Мы научились искать и фильтровать информацию с помощью регулярных выражений, а теперь давайте шагнем еще дальше, и научимся преобразовывать отфильтрованную информацию. Мы можем применить метод gsub, который в качестве параметров принимает само регулярное выражение, а также строку которая заменит найденное значение. Рассмотрим пример:

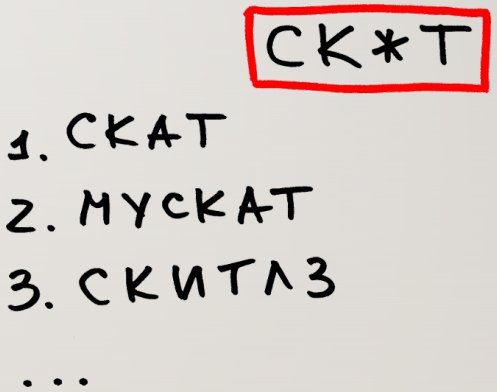
```
'I have an old car'.gsub('an old', 'a new')
=> "I have a new car"
```

```
'I have no money'.gsub(/no/, 'a heap of')
=> "I have a heap of money"
```

```
'I have a new car'.gsub(/a/, '1')
=> "I h1ve 1 new c1r"
```

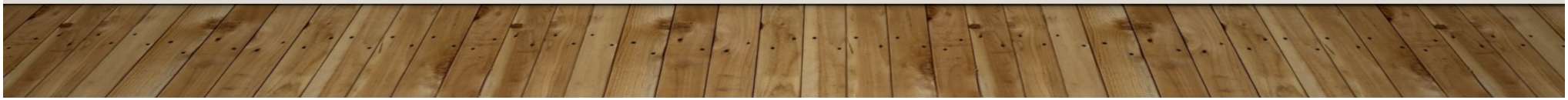
СОЗДАЕМ ИГРУ «УГАДАЙ СЛОВО»

- Начнем с постановки задачи. Условия игры следующие:
- Игрок видит определенный набор букв со звездочкой, и вместо звездочки ему необходимо подставить букву так, чтобы получилось существующее слово. Кроме того перед набором букв и после набора могут быть любые другие буквы в любом количестве. Человеку необходимо придумать как можно больше слов подходящих под представленный игрой шаблон. Для наглядности рассмотрим изображение:



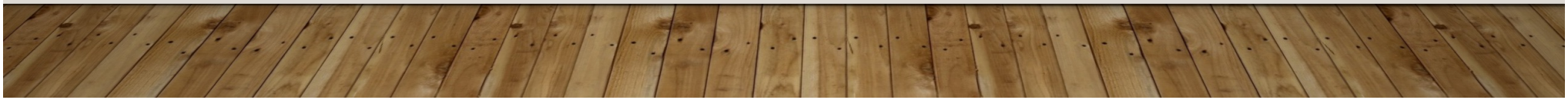
ПЛАН ЗАНЯТИЯ

- Кроме проверки соответствия шаблону, мы также будем проверять существует ли введенное пользователем слово в природе. А поможет нам в этом *Wiktionary* — Wiki-словарь.
- Весь код с подробнейшими комментариями берем из дополнительных материалов к уроку.



КОГДА СТОИТ ПРИМЕНЯТЬ РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

- Один из основных моментов, о котором всегда стоит помнить хорошему программисту — код должен быть легко читаемым. Некоторые регулярные выражения могут быть достаточно хитрыми, и не каждый разработчик сможет сходу разобраться, что сие выражение означает. Поэтому, если в определенной ситуации можно обойтись стандартными методами при решении задачи, то лучше воспользоваться ими. Не стоит усложнять себе и своим коллегам жизнь.
- В целом, регулярные выражения являются крайне полезным и мощным инструментом. Вы как Ruby-разработчик не обязаны быть мастером регулярных выражений, однако разбираться в них все же не будет лишним.
- Также вы будете лучше понимать, в какой ситуации стоить применять тот или иной паттерн.



ПРОВЕРЯЕМ EMAIL

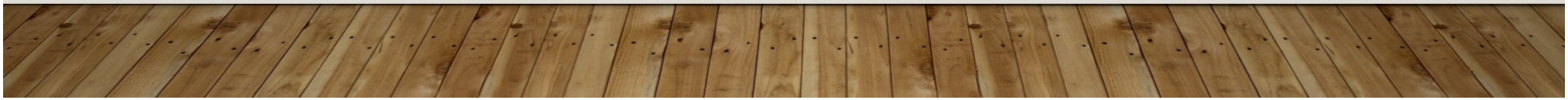
- Напишите программу, которая проверяет является ли введенный текст email-ом.
- **Например:**

```
$ ruby email.rb
Введите email:
email@gmail.com
Спасибо!

$ ruby email.rb
Введите email:
Какая-то фигня!
Это не email
```

ИЩЕМ В СТРОКЕ ХЭШТЕГИ

- Напишите программу, которая «вытаскивает» из строки, введенной пользователем хэштеги.
- Хэштегом мы считаем символ решетки и следующие за ним сколько угодно букв (как русских, так и латинских, как прописных, так и заглавных), цифр, знаков подчеркивания и минусов.
- Знаки препинания (запятая, точка, восклицательный и вопросительный знаки) и пробелы «рвут» хэштег.



ИЩЕМ В СТРОКЕ ХЭШТЕГИ

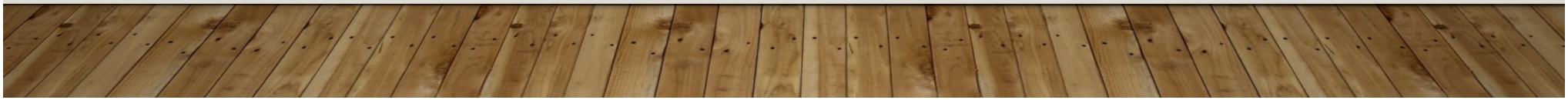
- **Например:**

```
Введите строку с хэштегами:
Будете у нас на #Колыме? Нет, уж лучше #вы_к_нам!

Нашли вот такие хэштеги: #Колыме, #вы_к_нам
```

ИЩЕМ В СТРОКЕ ХЭШТЕГИ. ПОДСКАЗКА

- Для поиска всех вхождений регулярного выражения в строке используйте метод строки scan.
- https://ruby-doc.org/core-2.4.0/String.html#method-i-scan
- А для составления хорошей и емкой регулярки почитайте документацию руби на класс Regexp:
- https://ruby-doc.org/core-2.4.0/Regexp.html



НА ТРИ ВЕСЕЛЫХ БУКВЫ...

- Напишите программу, которая читает текст из файла и считает количество слов из трех букв.
- **Подсказка:**
  - Используйте метод File.read, чтобы прочитать файл целиком.
  - Для проверки слова из трех букв можно использовать такое регулярное выражение: /^\S{3}$/

СПРАВОЧНАЯ ИНФОРМАЦИЯ

- Онлайн песочница для регулярок
- Памятка по регуляркам
- Примеры регулярок на руби
- Хороший читшит по регуляркам для разных языков
- Rubular — онлайн песочница, заточенная под Ruby
- Как исправить проблему с SSL сертификатами в руби программах на Windows
- Объяснение регулярок (eng)
- Туториал по регуляркам (eng)

СПАСИБО ЗА ВНИМАНИЕ!

Регулярные выражения в Ruby