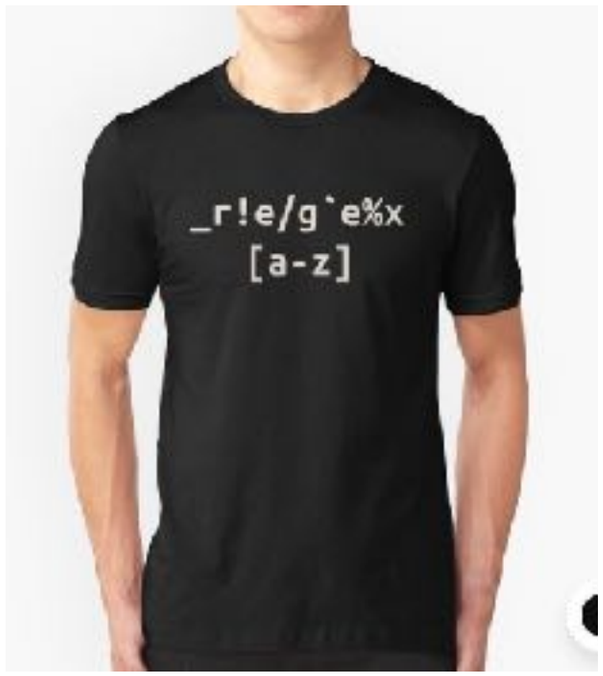
WTF IS REGEXP?

Kriszta Parrag
https://github.com/green-fox-academy/kitta11/

The purpose of my lighting talk is to give a sneak preview into the magical and weird world of regular expressions. I have picked this topic not because I am a master of them, but because I bumped into them many times during the first 5 weeks and always scared the shit out of me. So I decided to know them better. Hopefully we can be friends… or at least not enemies.

WHO ARE THEY? WHAT IS THEIR PURPOSE ON THE WORLD....

- purpose of their existence: providing a way to describe **patterns in string data**.
- habitat: they form a **small, separate language** that is part of many programming languages inc JavaScript
- occupation: powerful tool for inspecting and processing strings
- personality traits:
  - weird looking
  - extremely useful
  - cryptic syntax



So what is exactly a regular expression. A weird looking cryptic syntax? Yes. They form a small and separate language that is part of many other programming language including JavaScript. They are powerful tools for inspecting and processing strings especially because they provide a way to describe patterns in string data.

Aha. So what?

PROPERLY UNDERSTANDING REGULAR EXPRESSIONS WILL MAKE YOU A MORE EFFECTIVE PROGRAMMER.

pff....

Based on my research you can not be a programmer without understanding them. Or at least you wont be a good and effective one. I read many articles, guidelines and tried to collect the most important but not too advanced knowledge about them. Maybe in a 2nd session we can go deeper but lets start first with the basics.

AGENDA

1. String methods using regular expressions
2. Most popular built-in methods of the Regexp class
3. How to describe patterns with regular expressions
4. Using methods and cryptic patterns together

PLEASE FASTEN YOUR SEATBELTS BECAUSE

- REGULAR EXPRESSION IS A TYPE OF OBJECT (REGEXP)
- HOW TO CREATE:
  - using their constructor

written as a normal string

```
let myFirstRegExp = new RegExp('abc');
```

  - written as a literal value

written between slash characters

```
let mySecondRegExp = /abc/;
```

/ - forward slash

\ - backward slash

If you want to construct a regular expression you have two alternatives to do it. It can be done with the RegExp constructor or written as a literal value by enclosing a pattern in forward slash (/) characters. In the following 10 minutes I will use the second option and put the regexp always between these backward slashes.

The second notation, where the pattern appears between slash characters, treats backslashes somewhat differently. First, since a forward slash ends the pattern, we need to put a backslash before any forward slash that we want to be part of the pattern. In addition, backslashes that aren't part of special character codes (like \n) will be preserved, rather than ignored as they are in strings, and change the meaning of the pattern. Some characters, such as question marks and plus signs, have special meanings in regular expressions and must be preceded by a backslash if they are meant to represent the character itself.

1. STRING METHODS WITH REGEXPS

numerous STRING methods work with them and the regular expression format makes them smarter

SIMPLE STR.MATCH WITH AND WITHOUT REGEXP

```
let strTest: string = 'A dream you dream alone is only a dream.
A dream you dream together is reality';

console.log(strTest.match('dream'));
console.log(strTest.match(/dream/));
```

```
[ 'dream',
  index: 2,
  input: 'A dream you dream alone is only a dream. \nA dream you dream together is reality' ]
[ 'dream',
  index: 2,
  input: 'A dream you dream alone is only a dream. \nA dream you dream together is reality' ]
```

BOTH VERSION

- looks for the first match only
- returning an array with the searched phrase, its index, and the input string

We have a string. We want to use the MATCH method on this string with the word DREAM. Instead of typing the dream string we call this word in its regular expression format, between backslash brackets. In that format the method looks for the first match only. However returns with some useful information.
It returns with an array with that match and additional properties:
we get the index, the position of the match inside the string, and the input itself
If a part of the pattern is delimited by parentheses (...), then it becomes a separate element of the array.

STR.MATCH WITH REGEX – WITH G FLAG

```
let strTest: string = `A dream you dream alone is only a dream.
A dream you dream together is reality`;


console.log(strTest.match(/dream/ig));
// [ 'dream', 'dream', 'dream', 'dream', 'dream' ]
```

- returns an array of all matches (e.g. you can return quickly how many times the given expression can be found in the string)

However when there’s a "g" flag, then str.match returns an array of all matches. So you can for example easily tell how many times this word can be found in the given string.
If there are no matches, the call to match returns null

WHY STR.REPLACE METHOD WORKS BETTER WITH REGEXP?

```
let strTestReplace: string = `A dream you dream alone is only a dream.
A dream you dream together is reality`;

console.log(strTestReplace.replace('dream', 'nightmare'))
console.log(strTestReplace.replace(/dream/g, 'nightmare'))
```

```
.....................................
A nightmare you dream alone is only a dream.
A dream you dream together is reality
.....................................
A nightmare you nightmare alone is only a nightmare.
A nightmare you nightmare together is reality
.....................................
```

We already used the replace method with strings, but if we pass the parameter in a form of regular expression we have more opportunities ahead of us.
Similarly to the MATCH method REPLACE affect only the first match without the G flag. However G flag can be used only if the parameter is passed as a regexp. And tadadam without any for loop and further string manipulation all matches are replaced

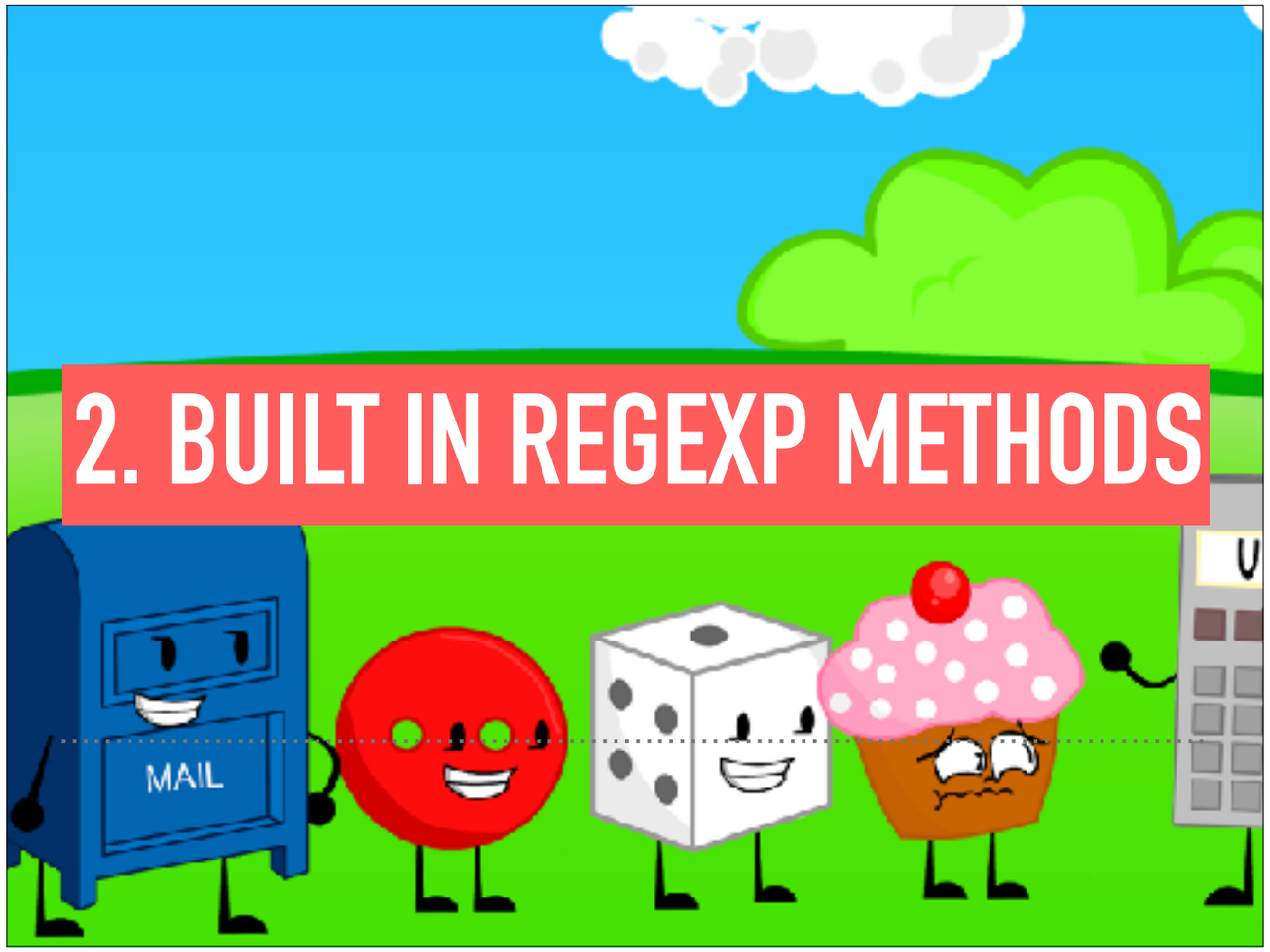
STR.REPLACE(/REGEXP/, FUNCTION) – WHAT???

```
//writing a function to show where is the match
function replacer(str, offset, s) {
  console.log(`Found "${str}" at position ${offset} in the provided text`);
  return str;

}
//calling replace method where the second param is the function
strTestReplace.replace(/dream/gi, replacer)
```

```
Found "dream" at position 2 in the provided text
Found "dream" at position 12 in the provided text
Found "dream" at position 34 in the provided text
Found "dream" at position 44 in the provided text
Found "dream" at position 54 in the provided text
```

But we should not stop here. Before that presentation I have never heard that you can pass a function also as a parameter to a string.replace method. Huhh.
For situations that require “smart” replacements, the second argument can be a function.
This function will be called for each match, and its result will be inserted as a replacement.
The case here demonstrate how we can print where we found these matches, under which index.



Ok but enough from the string methods. Regular expressions as a separate class have a lot of own method to use. lets see some example for them

REGEXP.TEST()

- If you pass it a string, it will return a Boolean telling you whether the string contains a match of the pattern in the expression.

```
let myRegexp = /kitta/;
let testString = 'The lighting talk by kitta is a little confusing'
console.log(myRegexp.test(testString)) //true
```

The most popular and the easiest is the test method, which is looking for any match and returns true/false whether he found it.
Let me demonstrate with a super simple example. I call the test method on myregexp and adding the test string variable to check the match. So it will check whether the testString has something like the kit word. This is not a rocket science. But this works also with more complicated regexp too. But lets talk about them a little later.

REGEXP.EXEC(STR) WITH G

- really similar to match but the way it returns the array is different

```
let strTestExec: string = `A dream you dream alone is only a dream.
A dream you drean together is reality`;

let myRegexp = /dream/ig;
console.log(myRegexp.exec(strTestExec))
console.log(myRegexp.exec(strTestExec))
console.log(myRegexp.exec(strTestExec))
console.log(myRegexp.exec(strTestExec))
console.log(myRegexp.exec(strTestExec))
console.log(myRegexp.exec(strTestExec))
```

An other super popular regular expression method is the exec.
It is like a match but with a special taste. It returns the first match and remembers the position after it in regexp.lastIndex property. The next call starts to search from regexp.lastIndex and returns the next match. If there are no more matches then regexp.exec returns null and regexp.lastIndex is set to 0.
I ran 6 times this method on our string and printed the result on the screen in order to demonstrate this amazing feature.

REGEXP.EXEC(STR) WITH G – LOOPING IT

- The main use case for regexp.exec is to find all matches in a loop and do something with the result. Let me show an example:

```
let strTestExec: string = `A dream you dream alone is only a dream.
A dream you dream together is reality`;

let myRegexp = /dream/ig;
let arrayToCollect: any [] = []
let result;

//The loop continues until regexp.exec returns null that means "no more matches".
while (result = myRegexp.exec(strTestExec)) {
  arrayToCollect.push(result.index)
}

console.log(arrayToCollect)
```

So the main use for this method is to find all matches in a loop and do something with the result. In this example I collected all indexes where the dream expression can be found into an array for further usage. So what I did is that I run a while loop, in which I called this exec method until it provided result and pushed the result index value into an other array.

1) {0,1,2} 0 or 1 or 2

R = 0 + 1 + 2

2) {∧, ab}

R = ∧ ab

3) {abb, a, b, bba} abb or a or b or

3. DESCRIBING PATTERNS WITH REGEXP

4) {∧, 0, 00, 000, ...}

.....

5) {1, 11, 111, 1111, ...}

LETS COMPLICATE IT FURTHER

The biggest advantage of regular expressions that they allow us to express more complicated patterns too.

I promise that we will finish this lighting talk soon, however I really want to mention one more important thing about these creatures. Their biggest power as I mentioned in the beginning is that they can describe patterns in strings and we can use these smart methods for these complex patterns too.

PATTERN 1: SETS OF CHARACTERS

- In a regular expression, putting a set of characters between [square brackets] makes that part of the expression match any of the characters between the brackets.

```
console.log(/[abcde]/.test('I like regexp')) //TRUE

console.log(/[0123456789]/.test('It is 2018')) //TRUE
```

- Within square brackets, a hyphen (-) between two characters can be used to indicate a range of characters.

```
console.log(/[a-e]/.test('I like regexp')) //TRUE

console.log(/[0-9]/.test('It is 2018')) //TRUE
```

for example we can define a set of characters by putting them among square brackets and run the method on this regular expression. First example check whether the string contains any of the ABCDE character set. Or the second one is testing the same for numbers.
Within the square brackets you can use the hyphen and define complex ranges.

PATTERN 1: SETS OF CHARACTERS – CONT.

- A number of common character groups have their own built-in shortcuts. Digits are one of them: \d means the same thing as [0-9].

`\d`	Any digit character
`\w`	An alphanumeric character ("word character")
`\s`	Any whitespace character (space, tab, newline, and similar)
`\D`	A character that is *not* a digit
`\W`	A nonalphanumeric character
`\S`	A nonwhitespace character
`.`	Any character except for newline

A number of common character groups have their own already built in shortcuts. For example d represents any digit, w represents any alphanumeric characters etc. I suggest to play with them at home to feel their taste.

PATTERN 1: SETS OF CHARACTERS – EXAMPLE

- Lets assume you need to match a special date format in your string like 01-30-2003 15:20

```
let dateTime = /\d\d-\d\d-\d\d\d\d \d\d:\d\d/;

console.log(dateTime.test("09-11-2018 23:20"));
// → true
console.log(dateTime.test("11-sep-1977 23:20"));
// → false
```

- to express that you want to match any character except the ones in the set—you can write a caret (^) character after the opening bracket.

```
let notPreferred = /[^ab]/
console.log(notPreferred.test('blablabla')) //true
console.log(notPreferred.test('bababa')) //false
```

So lets see a little more complicated and close to real life problem where the solution is to define a pattern. Lets assume we need to match a special date for in a given string. Where we have the month, the day , than the year, and the hour and minute.
So D represents digits, so we can describe this pattern like this.
If we created a pattern and we cant to match any other character except them we can write a caret . Like in the exam….

DO YOU REMEMBER THE 2ND TASK OF OUR EXAM?

```
let specialChar = /[^a-zA-Z]/gi
```

the “charet” means everything except them

a-z - all chars from a to z

PATTERN 2: REPEATING PARTS OF A PATTERN

- plus sign (+) after something in a regular expression: indicates that the element may be repeated more than once.

- star sign (*) after something in a regular expression: indicates repetition too however allows the pattern to match zero times

- question mark (?) makes a part of a pattern optional, meaning it may occur zero times or one time.

- braces {number} serve to indicate that a pattern should occur a precise number of times. It works with ranges too {2,4}

To make our syntax more cryptic we can use some characters to describe repetition in our string.
When you put a plus sign (+) after something in a regular expression, it indicates that the element may be repeated more than once. Thus, /\d+/matches one or more digit characters.
The star (*) has a similar meaning but also allows the pattern to match zero times.
question mark (?) makes a part of a pattern optional, meaning it may occur zero times or one time.
braces {number} serve to indicate that a pattern should occur a precise number of times. It works with ranges too {2,4}

PATTERN 2: REPEATING PARTS OF A PATTERN

- So the date and time format like 01-30-2003 15:20 can be described like this patter:

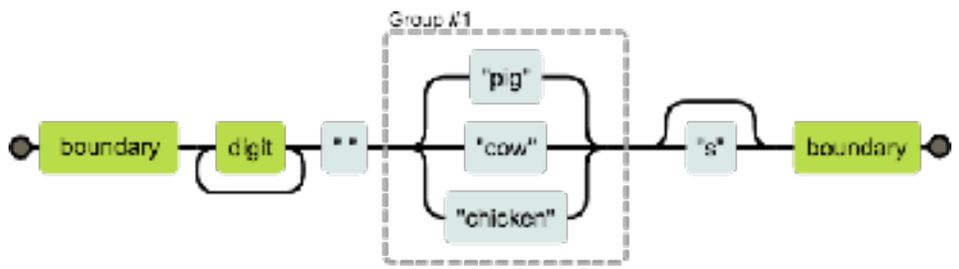
/\d{1,2}-\d{1,2}-\d{4} \d{1,2}:\d{2}/

digit with 1 or 2 times repetition	digit with 1 or 2 times repetition	digit with 4 times repetition	digit with 1 or 2 times repetition	digit with 2 times repetition

PATTERN 3: WORDS AND NUMBERS

- \b (word boundary)
- pipe character (|) denotes a choice between the pattern to its left and the pattern to its right
- Say we want to know whether a piece of text contains not only a number but a number followed by one of the words pig, cow, or chicken, or any of their plural forms.

```
let animalCount = /\b\d+ (pig|cow|chicken)s?\b/;
```

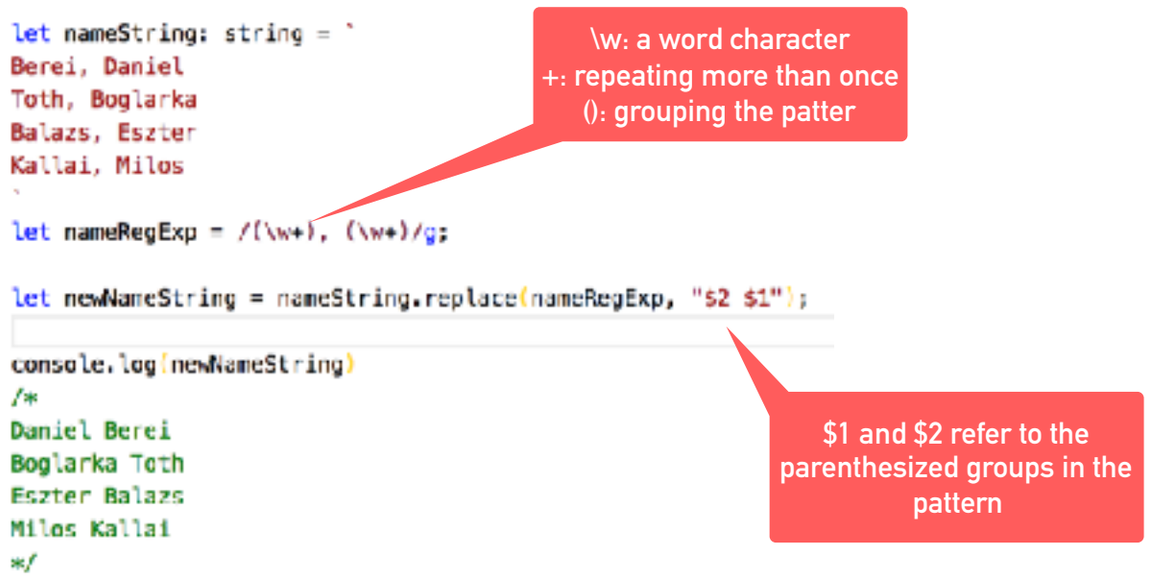


Our expression matches if we can find a path from the left side of the diagram to the right side.

FINALE: LETS MIX THIS TOGETHER

REPLACE + MATCHED PATTERNS

We have a long string with names in the following format: Lastname, Firstname. But we need Firstname Lastname format.



First lets mix the well known replace method but lets use it with some pattern. Lets assume we have a long string with names in the Lastname, Firstname format but we need Firstname Lastname without comma format. So what we can do? First of all lets describe our pattern in a form of a regular expression. So we have two parts both contains words characters so I can describe them with the w flag. Since we have two separate part we can use the parentheses to create groups which will be really useful in the method call. So using that pattern we call the replace on our string, where the first parameter is the pattern and the second parameter shows how I want to replace it.

REPLACE WITH FUNCTION + MATCHED PATTERNS 😱

We have a string in which we have a list of foods with the available quantity. Lets assume that every time we call a function the quantity is reduced is by one.

```
let myFridge = "1 lemon, 2 bananas, and 101 eggs”;
```

```
so where is the pattern?????
```

```
/(\d+) (\w+)/
```

group() containing numbers \d characters with repetition +

group() containing word \w characters with repetition +

The next one at first sight seems to be a nightmare. We have a string, like an inventory sentence about our fridge. A number followed by a string, then an other number ecetera. We want to write a function that can reduce these numbers with one every time we call it. So lets find first some patterns. we can identify a digit part where we can imagine some repetition so we use + and then a word part with repetition two.

REPLACE WITH FUNCTION + MATCHED PATTERNS 

Lets recall what we learnt regarding replace method when we are calling a function when there is a match….

The function is called with arguments `func(str, p1, p2, ..., pn, offset, s)`:

1. `str` – the match,
2. `p1, p2, ..., pn` – contents of parentheses (if there are any),
3. `offset` – position of the match,
4. `s` – the source string.

we will need this….

Before we jump to the solution lets recall what we learnt regarding replace method when a function is called with arguments. This function can contain as a parameter the str to be matched, the contents of parenthesised groups, the position and the string itself. What we need here is the second one, because we have numerous pattern group between parenthesis.

REPLACE WITH FUNCTION + MATCHED PATTERNS 😱

So lets put together the function and call it within the replace method.

```
let myFridge = "1 lemon, 2 bananas, and 101 eggs";

function minusOne(match, $1, $2) {
  $1 = Number($1) - 1;
```

$1 and $2 refer to the two group inside the pattern (we can call them whatever we want)
$1(the number) - (\d+)
$2(the item name) - (\w+)

so lets put together everything we have


I'M SORRY
FORGIVE ME!