What is RegEx?

- RegEx stands for Regular Expressions
- A way to create a filter pattern
- What if my data is not a database, but unstructured data?
- Uses:
  - Finding specific text in unstructured data
  - Finding the right item
  - Verify structure
  - Do replacements
- Practical Example:
  - Validating input fields (websites, apps, etc.)
    - Passwords, emails…
  - Identifying spam websites (ScanNet)

RegEx in DM

→ Why is RegEx a part of Data Management?

→ Relates to getting data (just as we do from databases)

→ Valuable in relation to unstructured data

RegEx Example

```
/([A-Z])\w+/g
```

Text | Tests NEW | 29 matches (1.0ms)

```
RegExr was created by gskinner.com, and is proudly hosted by Media Temple.

Edit the Expression & Text to see matches. Roll over matches or the expression for details. PCRE & JavaScript flavors of RegEx are supported. Validate
your expression with Tests mode.

The side bar includes a Cheatsheet, full Reference, and Help. You can also Save & Share with the Community, and view patterns you create or favorite in
My Patterns.

Explore results with the Tools below. Replace & List output custom results. Details lists capture groups. Explain describes your expression in plain
English.
```

See tool here: https://regexr.com – examples today will be in VS Code

Anchors

Anchors	
^	Matches at the start of string or start of line if multi-line mode is enabled. Many regex implementations have multi-line mode enabled by default.
$	Matches at the end of string or end of line if multi-line mode is enabled. Many regex implementations have multi-line mode enabled by default.
\A	Matches at the start of the search string.
\Z	Matches at the end of the search string, or before a newline at the end of the string.
\z	Matches at the end of the search string.
\b	Matches at word boundaries.
\B	Matches anywhere but word boundaries.

Character Classes

Character Classes *Can also be used in bracket expressions.*	
`.`	Matches any character except newline. Will also match newline if single-line mode is enabled.
`\s`	Matches white space characters.
`\S`	Matches anything but white space characters.
`\d`	Matches digits. Equivalent to [0-9].
`\D`	Matches anything but digits. Equivalent to [^0-9].
`\w`	Matches letters, digits and underscores. Equivalent to [A-Za-z0-9_].
`\W`	Matches anything but letters, digits and underscores. Equivalent to [^A-Za-z0-9_].
`\xff`	Matches ASCII hexadecimal character ff.
`\x{ffff}`	Matches UTF-8 hexadecimal character ffff.
`\cA`	Matches ASCII control character ^A. Control characters are case insensitive.
`\132`	Matches ASCII octal character 132.

Groups

Groups	
`(foo|bar)`	Matches pattern foo or bar.
`(foo)`	Define a group (or sub-pattern) consisting of pattern foo. Matches within the group can be referenced in a replacement using a backreference.
`(?<foo>bar)`	Define a named group named "foo" consisting of pattern bar. Matches within the group can be referenced in a replacement using the backreference $foo.
`(?:foo)`	Define a passive group consisting of pattern foo. Passive groups cannot be referenced in a replacement using a backreference.
`(?>foo+)bar`	Define an atomic group consisting of pattern foo+. Once foo+ has been matched, the regex engine will not try to find other variable length matches of foo+ in order to find a match followed by a match of bar. Atomic groups may be used for performance reasons.

Bracket Expressions

Bracket Expressions	
`[adf]`	Matches characters a or d or f.
`[^adf]`	Matches anything but characters a, d and f.
`[a-f]`	Match any lowercase letter between a and f inclusive.
`[A-F]`	Match any uppercase letter between A and F inclusive.
`[0-9]`	Match any digit between 0 and 9 inclusive. Does not support using numbers larger than 9, such as [10-20].

Quantifiers

- RegExs are greedy!
- {min, max} / {min,} / {, max} / {exact}
- ? -> {0,1}
- + -> {1,0}
- * -> {0,}

Quantifiers	
*	0 or more. Matches will be as large as possible.
*?	0 or more, lazy. Matches will be as small as possible.
+	1 or more. Matches will be as large as possible.
+?	1 or more, lazy. Matches will be as small as possible.
?	0 or 1. Matches will be as large as possible.
??	0 or 1, lazy. Matches will be as small as possible.
{2}	2 exactly.
{2,}	2 or more. Matches will be as large as possible.
{2,}?	2 or more, lazy. Matches will be as small as possible.
{2,4}	2, 3 or 4. Matches will be as large as possible.
{2,4}?	2, 3 or 4, lazy. Matches will be as small as possible.

Special Characters

Special Characters	
\	Escape character. Any metacharacter to be interpreted literally must be escaped. For example, \? matches literal ?. \\ matches literal \.
\n	Matches newline.
\t	Matches tab.
\r	Matches carriage return.
\v	Matches form feed/page break.

Assertions

Assertions	
`foo(?=bar)`	Lookahead assertion. The pattern foo will only match if followed by a match of pattern bar.
`foo(?!bar)`	Negative lookahead assertion. The pattern foo will only match if not followed by a match of pattern bar.
`(?<=foo)bar`	Lookbehind assertion. The pattern bar will only match if preceded by a match of pattern foo.
`(?<!foo)bar`	Negative lookbehind assertion. The pattern bar will only match if not preceded by a match of pattern foo.

POSIX Character Classes

POSIX Character Classes Must be used in bracket expressions, e.g. [a-z[:upper:]]	
`[:upper:]`	Matches uppercase letters. Equivalent to A-Z.
`[:lower:]`	Matches lowercase letters. Equivalent to a-z.
`[:alpha:]`	Matches letters. Equivalent to A-Za-z.
`[:alnum:]`	Matches letters and digits. Equivalent to A-Za-z0-9.
`[:ascii:]`	Matches ASCII characters. Equivalent to \x00-\x7f.
`[:word:]`	Matches letters, digits and underscores. Equivalent to \w.
`[:digit:]`	Matches digits. Equivalent to 0-9.
`[:xdigit:]`	Matches characters that can be used in hexadecimal codes. Equivalent to A-Fa-f0-9.
`[:punct:]`	Matches punctuation.
`[:blank:]`	Matches space and tab. Equivalent to [\t].
`[:space:]`	Matches space, tab and newline. Equivalent to \s.
`[:cntrl:]`	Matches control characters. Equivalent to [\x00-\x1F\x7F].
`[:graph:]`	Matches printed characters. Equivalent to [\x21-\x7E].
`[:print:]`	Matches printed characters and spaces. Equivalent to [\x21-\x7E].

Replacement Backreferences

Replacement Backreferences Used in replacements	
`$3 or \3 or`	Matched string within the third non-passive group.
`$0 or $& or \0`	Entire matched string.
`$foo` `${foo}`	Matched string within the group named "foo".

Case Modifiers

Modifiers May be grouped together, e.g. (?ixm)	
`(?i)`	Case insensitive mode. Make the remainder of the pattern or sub-pattern case insensitive.
`(?m)`	Multi-line mode. Make $ and ^ in the remainder of the pattern or subpattern match before/after newline.
`(?s)`	Single-line mode. Make the . (dot) in the remainder of the pattern or subpattern match newline.
`(?x)`	Free spacing mode. Ignore white space in the remainder of the pattern or subpattern.

Recursive Backreferences

Recursive Backreferences Used in patterns to reference captured text or sub-patterns from a capture group. Only available in some regex implementations.	
`\3 or \k<3>`	Re-match the text previously matched within the third non-passive group.
`\k<foo>`	Re-match the text previously matched within the group named "foo".
`\g<3>`	Re-execute the subpattern within the third non-passive group.
`\g<foo>`	Re-execute the sub-pattern within the group named "foo".

Expression Flags

- Placed after the /expression/
- (bad) JavaScript Example: /\w+@\w+.\w{2,3}/g
  - Not all languages use /expression/g flags
  - See java example on next page for for case insensitive example

Expression Flags	
i	With this flag the search is case-insensitive: no difference between A and a.
g	With this flag the search looks for all matches, without it – only the first match is returned.
m	
s	Enables “dotall” mode, that allows a dot . to match newline character \n.
u	Enables full Unicode support. The flag enables correct processing of surrogate pairs.
y	“Sticky” mode: searching at the exact position in the text.

RegEx examples in Java

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("w3schools", Pattern.CASE_INSENSITIVE);
    Matcher matcher = pattern.matcher("Visit W3Schools!");
    boolean matchFound = matcher.find();
    if(matchFound) {
      System.out.println("Match found");
    } else {
      System.out.println("Match not found");
    }
  }
}
// Outputs Match found
```

Se more here: https://www.w3schools.com/java/java_regex.asp