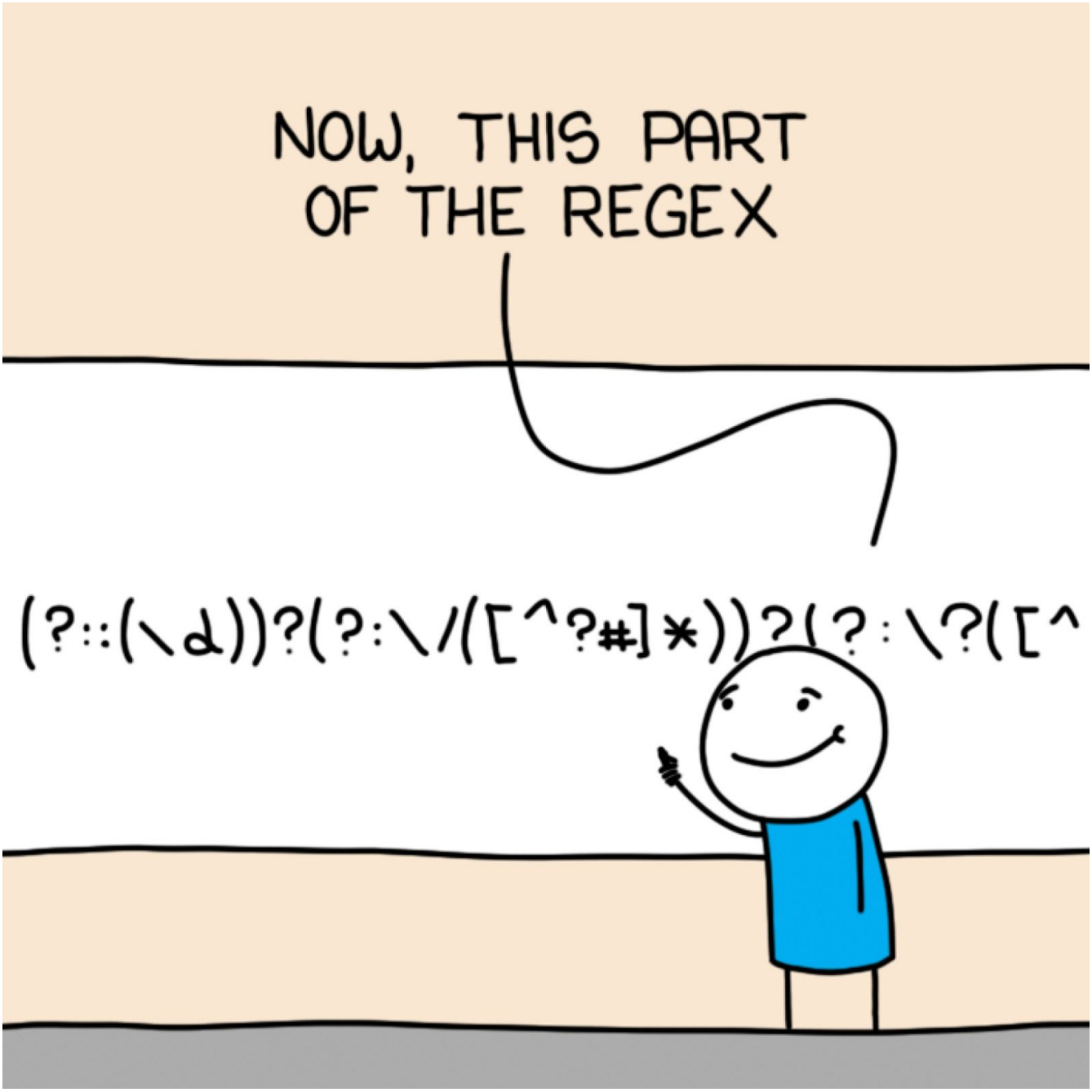
Regulární výrazy

Vtip



- Regex Humor

Co jsou regulární výrazy?

- Velmi často používané konstrukce pro hledání v řetězcích.
- Například pomocí nich lze ověřit, že řetězec by mohl být validní e-mailová adresa.
- Používají se tedy například pro validaci vstupů.
- Zvládnutí aspoň an základní úrovni je skoro nezbytností - často i pro neprogramátory.

- Bohužel v česko-slovenském (a jiném) prostředí s diakritikou přináší zrovna javová implementace potíže.
- Povšechné, ne javové, info k regex je také na https://www.regularnivyrazy.info/

Regex v Javě a jinde

- Fakticky jsou to výrazy definující tzv. regulární (formální) gramatiky dle Chomského hierarchie.
- Valná většina pg. jazyků umí nějakou formu interpretace regulárních výrazů.
- Javové regulární výrazy (slangově regex) jsou syntakticky podobné jako např. v Perlu.
- Mírně, nikoli principiálně, se liší od regexů v `grep`, `Python`, `PHP`, `awk`, `sed`.

Podpora regex v Javě

- Balík `java.util.regex` se skládá ze tří tříd:

  Objekt `Pattern`

  je zkompilovaný regex. Chceme-li jej vytvořit, musíme nejprve zavolat jednu z veřejných statických metod. Tyto metody přijímají jako první argument regulární výraz ve formě řetězce a vracejí ho zkompilovaný v objektu `Pattern`.

  Objekt `Matcher`

  interpretuje `Pattern` a provádí operace porovnání se vstupním řetězcem typu `String`. Objekt `Matcher` získáme voláním metody `matcher` na objektu `Pattern`.

  Objekt `PatternSyntaxException`

  je nehlídaná výjimka, která indikuje syntaktickou chybu v regulárním výrazu.

Práce s regex

```
// string with a regex
String regex = ".";
// compiled regex pattern
Pattern pattern = Pattern.compile(regex);
// pattern used to match a given text
Matcher matcher = pattern.matcher(text);
// may match 0 or more time
int matches = 0;
while (matcher.find()) {
   matches++;
}
```

Jednou zkompiluj, vícekrát použij

```
private static final Pattern keyValuePattern
   = Pattern.compile("^([^\\:]+)\\:(.*)");

private String[] keyValue(String text) {
   Matcher matcher = pattern.matcher(text);
   return matcher.find();
}
```

Příklad regexu

- Je zkoumaný řetězec názvem javového souboru s názvem začínajícím na `Exx`, kde *xx* jsou dvě číslice?
- Regex bude `^E\d\d.*\.java`.
- `^E` znamená, že zkoumaný řetězec začíná na `E`,
- následují právě dvě číslice dané `\d\d`, kde `\d` je jedna číslice 0..9,
- dále `.*` je libovolně dlouhá (i prázdná) sekvence libovolných znaků
- `\.` je znak tečka,
- `java` je sekvence čtyř znaků "java".

Metaznaky

- `[]` - vymezení třídy znaků - v závorkách je množina znaků, výraz odpovídá kterémukoli ze znaků ze třídy
- v Javě musíme backslash v řetězcovém literálu zdvojit
- `"\\t"` - regulární výraz pro tabelátor zapsaný formou řetězce

Třídy znaků (*classes*)

`[abc]`

: odpovídá libovolnému ze znaků uvedených v `[]`, tedy v řetězci `cba` je tento regulární výraz obsažen hned 3x.

`[a-z]`

: všechna malá písmena (ASCII, bez diakritiky).

`[^a-z]`

: všechno kromě malých písmen z ASCII, bez diakritiky.

`[^abc]`

neguje třídu znaků, tzn. odpovídá libovolnému znaku mimo znaků uvedených v `[]`, tedy v řetězci `xyzcba` je tento regulární výraz obsažen také 3x.

Jednotlivé speciální znaky

`\t`

tabulátor

`\r`

CR, konec řádku

`\n`

LF linefeed, nový řádek

`\v`

vertikální tabulátor

`\f`

odstránkování

Předdefinované třídy znaků v regex

- Regexy tradičně používaly třídy pro bílé znaky, znaky identifikátorů, písmena, číslice atd.

`.`

jakýkoli znak kromě ukončení řádků, tzn. `\n`, `\r`, `\u2028` a `\u2029`

`\s`

libovolný bílý znak (mezera, tab, newline, cr), `\S` libovolný nebílý znak

`\w`

libovolný znak slova (word character) - spolehlivě funguje pro znaky v `[A-Za-z0-9_]`, pro češtinu či slovenštinu nefunguje správně, `\W` libovolný jiný znak než slova (non-word character)

`\d`

číslice 0..9 (jen tyto naše, arabské), `\D` cokoli mimo ně

Pozice regulárního výrazu v řetězci

`^abc`

odpovídá sekvenci znaků `abc` na začátku prohledávaného řetězce: `abcdef` ano, ale `defabc` nikoli.

`abc$`

odpovídá sekvenci znaků `abc` na konci prohledávaného řetězce: `abcdef` ne, ale `defabc` ano.

Skupiny pro zachycení

- Neboli *matching groups*.
- Vymezují se závorkami `()`.
- Používají se, když volání `matcher.find() == true` pak `matcher.group(n)` je n-tá zachycená skupina - jsou číslované od 1.
- Celý detekovaný výraz je `matcher.group(0)`.
- Třeba `(abc)def` zachytí sekvenci `abc` v řetězci `abcdefghijk` jako `group(1)`,
- zatímco `group(0)` nebo `group()` je celý regex `(abc)def`

Příklad skupin

```
if(matcher.matches())
   // if matches then the groups are available
   String key = matcher.group(1);
   String value = matcher.group(2);
   // group(0) would capture the entire text matching the regex
```

UNICODE v Javě

- Bohužel javová interpretace regexů pro novější kategorie znaků v UNICODE není správná.
- Selhávat to bude v abecedách nověji přijatých do UNICODE.
- Blíže viz Java's Regex Unicode Problems
- V tomto článku je i workaround