Zaawansowane programowanie w Javie

Studia zaoczne – Wykład 7

dr hab. Andrzej Zbrzezny, profesor UJD

Katedra Matematyki i Informatyki
Uniwersytet Jana Długosza w Częstochowie

25 maja 2024

Zalecana Literatura

Literatura podstawowa

- Cay Horstmann.
  Java. Techniki zaawansowane. Wydanie XI.
  Wydawnictwo Helion. Gliwice, listopad 2019.
  Wyrażenia regularne opisane w podrozdziale 2.7.
  Cały rozdział 2 do pobrania ze strony:
  https://pdf.helion.pl/jatz11/jatz11.pdf
- Cay Horstmann.
  Java 9. Przewodnik doświadczonego programisty. Wydanie II.
  Wydawnictwo Helion. Gliwice, maj 2018.
- https://docs.oracle.com/javase/tutorial/essential/regex/

Zalecana Literatura

Literatura uzupełniająca

- https://www.samouczekprogramisty.pl/wyrazenia-regularne-w-jezyku-java/
- https://www.samouczekprogramisty.pl/wyrazenia-regularne-czesc-2/

Wyrażenia regularne

Wprowadzenie

- Wyrażenia regularne to sposób opisywania zbioru ciągów znaków na podstawie cech wspólnych dla każdego ciągu w tym zbiorze.
- Można ich używać do wyszukiwania, edycji lub manipulacji tekstem i danymi.
- Aby tworzyć wyrażenia regularne, należy poznać specyficzną składnię – taką, która wykracza poza normalną składnię języka programowania Java.
- Wyrażenia regularne różnią się stopniem złożoności, ale gdy zrozumie się podstawy ich konstrukcji, będzie można rozszyfrować (lub utworzyć) dowolne wyrażenie regularne.

Wyrażenia regularne

Wprowadzenie

- Załóżmy na przykład, że chcemy odnaleźć odnośniki w pliku HTML. Musimy szukać ciągów znaków pasujących do wzorca `<a href="...">`.
- Jednak to nie wszystko - mogą pojawić się dodatkowe odstępy lub URL może być zamknięty w pojedynczych cudzysłowach.
- Wyrażenia regularne udostępniają precyzyjną składnię pozwalającą określać, jakie ciągi liter są poprawnym dopasowaniem.
- Na kolejnych slajdach zobaczymy składnię wyrażeń regularnych wykorzystywaną w Java API i dowiemy się, w jaki sposób korzystać z wyrażeń regularnych.

Wyrażenia regularne w Javie

Pakiet `java.util.regex`

- Pakiet `java.util.regex` składa się głównie z trzech klas: `Pattern`, `Matcher` oraz `PatternSyntaxException`.
- Klasa `Pattern` nie udostępnia żadnych publicznych konstruktorów.
- Obiekt klasy `Pattern` jest skompilowaną reprezentacją wyrażenia regularnego.
- Aby utworzyć wzorzec, należy najpierw wywołać jedną z dwóch publicznych statycznych metod `Pattern.compile`, która zwróci obiekt klasy `Pattern`.
- Obie te metody te przyjmują jako pierwszy argument wyrażenie regularne, natomiast druga jako swój drugi argument przyjmuje wartość całkowitą `flags`.

Wyrażenia regularne w Javie

Pakiet `java.util.regex`

- Obiekt klasy `Matcher` implementującej interfejs `MatcherResult` jest mechanizmem, który interpretuje wzorzec i wykonuje operacje dopasowania do łańcucha wejściowego.
- Podobnie jak klasa `Pattern`, także klasa `Matcher` nie definiuje żadnych publicznych konstruktorów.
- Obiekt `Matcher` uzyskuje się przez wywołanie metody `matcher` na obiekcie klasy `Pattern`.
- Obiekt `PatternSyntaxException` jest niekontrolowanym wyjątkiem, który wskazuje na błąd składni we wzorcu wyrażenia regularnego.
- Przed szczegółowym zapoznaniem się z klasami pakietu `java.util.regex` trzeba najpierw zrozumieć, jak właściwie konstruowane są wyrażenia regularne w Javie.

Wyrażenia regularne

Składnia wyrażeń regularnych

- W wyrażeniach regularnych wszystkie znaki oprócz wymienionych poniżej 12 znaków zastrzeżonych oznaczają takie same znaki:
  `. * + ? { | ( ) [ \ ^ $`
- Na przykład wyrażenie regularne `Java` pasuje jedynie do ciągu znaków `Java`.
- Znak `.` jest dopasowywany do dowolnego znaku. Przykładowo, `.a.a` zostanie dopasowane zarówno do `Java`, jak i `data`.
- Znak `*` oznacza, że poprzedzające konstrukcje mogą być powtórzone 0 lub więcej razy; dla znaku `+` jest to 1 lub więcej razy.
- Przyrostek `?` oznacza, że konstrukcja jest opcjonalna (0 lub 1 raz). Przykładowo, `be+s?` dopasowuje się do `be` , `bee` oraz `bees`.
- Można określić inne liczby powtórzeń za pomocą `{ }` – szczegółowe informacje znajdują się w tabeli `2.6` z książki C. Horstmanna `Java. Techniki zaawansowane`.

Wyrażenia regularne

Składnia wyrażeń regularnych

- Klasa znaków (ang. character class) jest zestawem alternatywnych znaków zamkniętych w nawiasach, takim jak `[Jj]`, `[0-9]`, `[A-Za-z]` czy `[^0-9]`.
- Wewnątrz klasy znaków znak `-` służy do opisu zakresu (wszystkich znaków, których wartości Unicode leżą pomiędzy dwoma krańcowymi wartościami).
- Jednak znak `-` będący pierwszym lub ostatnim znakiem klasy znaków reprezentuje sam siebie.
- Znak `^` jako pierwszy znak w klasie znaków oznacza dopełnienie (wszystkie znaki oprócz określonych w klasie).
- Istnieje wiele predefiniowanych klas znaków, takich jak `\d` (cyfry) czy `\p{Sc}` (oznaczenia walut Unicode) – szczegółowe informacje znajdują się w tabeli `2.6` z książki C. Horstmanna `Java. Techniki zaawansowane`.

Wyrażenia regularne

Składnia wyrażeń regularnych

- Znaki `^` oraz `$` są dopasowywane do początku i końca danych wejściowych.
- Istnieją dwa sposoby wymuszenia, aby metaznak ze zbioru
  `. * + ? { | ( ) [ \ ^ $`
  był traktowany jak zwykły znak:
  - można poprzedzić metaznak odwrotnym ukośnikiem, lub
  - zawrzeć go w `\Q` (rozpoczynającym cytat) i `\E` (kończącym cytat).
- Przy użyciu tej techniki znaki `\Q` i `\E` można umieszczać w dowolnym miejscu wyrażenia, pod warunkiem, że `\Q` jest na początku.
- Wewnątrz klasy znaków należy poprzedzać znakiem `\` jedynie znaki `[` oraz `\`, oczywiście uwzględniając położenie znaków `\ - ^`.
- Przykładowo, klasa `[]^-]` zawiera wszystkie trzy wymienione znaki.

Wyrażenia regularne

Klasy znaków

Klasa znaków	Dopasuj
`.`	dowolny znak z wyjątkiem znaku nowej linii
`\d`	dowolną cyfrę 0–9
`\D`	dowolny znak niebędący cyfrą
`\s`	dowolny biały znak, w tym `\t`, `\n` i `\r` oraz znak spacji
`\S`	dowolny znak niebędący białym znakiem
`\w`	dowolny znak z klasy `[a-zA-Z0-9_]`
`\W`	dowolny znak nienależący do klasy `\w`
`\p{L}` lub `\pL`	dowolny alfanumeryczny znak Unicode'u
`\P{L}` lub `\PL`	dowolny znak niebędący alfanumerykiem
`\t`, `\n`, `\r`	znak tabulatora, nowej linii, powrotu karetki
`[...]`	jeden znak spośród zbioru znaków
`[^...]`	jeden znak spoza zbioru znaków

Wyrażenia regularne

Przykłady (Klasy znaków)

Klasa znaków	Dopasuj
`[abc]`	`a`, `b`, or `c` (prosta klasa)
`[^abc]`	Dowolny znak z wyjątkiem `a`, `b` lub `c` (negacja)
`[a-zA-Z]`	od `a` `z` lub od `A` do `Z` włącznie (zakres)
`[a-d[m-p]]`	od a do d lub od m dp p: `[a-dm-p]` (suma)
`[a-z&&[def]]`	`d`, `e`, or `f` (przecięcie, inaczej: część wspólna)
`[a-z&&[^bc]]`	od `a` do `z` bez `b` i `c`: `[ad-z]` (odejmowanie)
`[a-z&&[^m-p]]`	`[a-z]` bez `[m-p]`: `[a-lq-z]` (odejmowanie)

Wyrażenia regularne

Odnajdywanie jednego lub wszystkich dopasowań

- Ogólnie mówiąc, istnieją dwa sposoby korzystania z wyrażeń regularnych:
  - albo chcemy ustalić, czy ciąg znaków pasuje do wyrażenia,
  - albo chcemy odnaleźć wszystkie wystąpienia wyrażenia regularnego w ciągu znaków.
- W pierwszym przypadku należy skorzystać ze statycznej metody `matches` :

```
String regex = "[+-]?\\d+";
CharSequence dane = ...;
if (Pattern.matches(regex, dane)) {
    ...
}
```

Wyrażenia regularne

Odnajdywanie jednego lub wszystkich dopasowań

- Jeżeli musimy wykorzystać to samo wyrażenie regularne wiele razy, bardziej wydajne będzie skompilowanie go.
- Następnie należy utworzyć obiekt klasy `Matcher` dla każdych danych wejściowych:

```
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(dane);
if (matcher.matches()) {
    ...
}
```

- Jeżeli uda się dopasować ciąg znaków, można pobrać lokalizację dopasowanych grup – zobaczymy to w kolejnej części.

Wyrażenia regularne

Odnajdywanie jednego lub wszystkich dopasowań

- Jeżeli chcemy dopasować elementy kolekcji lub strumienia, należy przekształcić wzorzec w predykat:

```
Stream<String> ciągi = ...;
Stream<String> wyniki =
    ciągi.filter(pattern.asPredicate());
```

W wynikach znajdują się wszystkie ciągi znaków pasujące do wyrażeń regularnych.

Wyrażenia regularne

Odnajdywanie jednego lub wszystkich dopasowań

- Rozważmy też drugi sposób użycia – wyszukiwanie wszystkich wystąpień wyrażenia regularnego we wskazanym ciągu znaków.
- Należy użyć następującej pętli:

```
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(input);
while (matcher.find()) {
    String match = matcher.group();
    ...
}
```

- W ten sposób można przetworzyć kolejno wszystkie dopasowania.
- W kolejnej części zobaczymy, w jaki sposób można zająć się wszystkimi dopasowaniami równocześnie.

Wyrażenia regularne w Javie

Odnajdywanie jednego lub wszystkich dopasowań

- Do badania składni wyrażeń regularnych można użyć programu `RegexTestHarness.java` lub jego zmodyfikowanej wersji `RegexTesting.java`.

Przykłady (Testowanie składni wyrażeń regularnych)

- Program `regex/RegexTestHarness.java`
- Program `regex/RegexTesting.java`

Przykład (Odnajdywanie jednego lub wszystkich dopasowań)

- Program `regex/FindingOneOrAllMatches.java`

Wyrażenia regularne

Kwantyfikatory

Kwantyfikator			Dopasuj poprzedzający element
zachłanny	oporny	zaborczy	
`?`	`??`	`?+`	`0` lub `1` raz
`*`	`*?`	`*+`	`0` lub więcej razy
`+`	`+?`	`++`	`1` lub więcej razy
`{m}`	`{m}?`	`{m}+`	dokładnie `m` razy
`{m,}`	`{m,}?`	`{m,}+`	co najmniej `m` razy
`{m,n}`	`{m,n}?`	`{m,n}+`	od `m` do `n` razy

Granice słowa

Asercja	Dopasuj
`\b`	granicę słowa w łańcuchu znaków

Wyrażenia regularne

Kwantyfikatory zachłanne, oporne i zaborcze

- Istnieją subtelne różnice między kwantyfikatorami zachłannymi (ang.greedy), opornymi (ang. reluctant) i zaborczymi (ang. possesive).
- Kwantyfikatory zachłanne są nazywane „zachłannymi”, ponieważ zmuszają algorytm dopasowujący do wczytania lub całego łańcucha wejściowego przed podjęciem pierwszej próby dopasowania.
- Jeżeli pierwsza próba dopasowania (całego łańcucha wejściowego) nie powiedzie się, algorytm dopasowujący cofa łańcuch wejściowy o jeden znak i próbuje ponownie, powtarzając ten proces aż do znalezienia dopasowania lub do momentu, gdy nie pozostanie już żadnych znaków do cofnięcia.

Wyrażenia regularne

Kwantyfikatory zachłanne, oporne i zaborcze

- W zależności od kwantyfikatora użytego w wyrażeniu, ostatnią rzeczą, dla której zostanie podjęta próba dopasowania, jest 1 lub 0 znaków.
- Kwantyfikatory oporne przyjmują odwrotne podejście: zaczynają od początku łańcucha wejściowego, a następnie niechętnie sprawdzają jeden znak po drugim w poszukiwaniu dopasowania.
- Ostatnią rzeczą, którą próbują, jest cały łańcuch wejściowy.
- Wreszcie, kwantyfikatory zaborcze zawsze analizują cały łańcuch wejściowy, próbując raz (i tylko raz) znaleźć dopasowanie.
- W przeciwieństwie do kwantyfikatorów zachłannych, kwantyfikatory zaborcze nigdy się nie wycofują, nawet jeśli pozwoliłoby to na udane dopasowanie.

Wyrażenia regularne

Przykład (Kwantyfikatory zachłanne, oporne i zaborcze)

```
Enter your regex: .*foo
Enter input string to search: xfooxxxfoo
I found the text "xfooxxxfoo" starting at 0 and ending at 10.

Enter your regex: .*?foo
Enter input string to search: xfooxxxfoo
I found the text "xfoo" starting at 0 and ending at 4.
I found the text "xxxfoo" starting at 4 and ending at 10.

Enter your regex: .*+foo
Enter input string to search: xfooxxxfoo
No match found.
```

Przykład

- Program `regex/RegexTestHarness.java`

Wyrażenia regularne

Testowanie wyrażeń regularnych

- Do badania konstrukcji wyrażeń regularnych obsługiwanych przez pakiet `java.util.regex` można używać statycznej funkcji funkcji `reth` z programu `regex/Reth.java`
- Po skompilowaniu tego programu, jego uruchomienie wyświetli krótką informację o sposobie jego użycia w narzędziu `jshell`.
- Użycie tego narzędzia testowego jest opcjonalne, ale może okazać się wygodne podczas poznawania przypadków testowych omawianych na poprzednim i na dzisiejszym wykładzie.

Wyrażenia regularne

Grupy

- Grupy przechwytujące to sposób traktowania wielu znaków jako pojedynczej jednostki. Tworzy się je przez umieszczenie znaków, które mają być zgrupowane, wewnątrz zestawu nawiasów.
- Na przykład wyrażenie regularne `(dog)` tworzy pojedynczą grupę zawierającą litery `"d"`, `"o"` i `"g"`.
- Część łańcucha wejściowego, która pasuje do grupy przechwytującej, zostanie zapisana w pamięci w celu późniejszego przywołania za pomocą odsyłaczy wstecznych.
- Grupy przechwytujące są numerowane przez liczenie ich nawiasów otwierających od lewej do prawej. Przykładowo, w wyrażeniu `((A)(B(C)))` istnieją cztery takie grupy:
  1. `((A)(B(C)))`
  2. `(A)`
  3. `(B(C))`
  4. `(C)`

Wyrażenia regularne

Grupy

- Aby dowiedzieć się, ile grup występuje w wyrażeniu, należy wywołać metodę `groupCount` na obiekcie klasy `Matcher`.
- Metoda `groupCount` zwraca liczbę całkowitą określającą liczbę grup przechwytujących obecnych we wzorcu dopasowującym.
- W poprzednim przykładzie metoda `groupCount` zwróci liczbę `4`, co oznacza, że wzorzec zawiera `4` grupy przechwytujące.
- Istnieje również specjalna grupa, grupa `0`, która zawsze reprezentuje całe wyrażenie.
- Ta grupa nie jest uwzględniana w sumie podawanej przez `groupCount`.
- Grupy rozpoczynające się od znaków `(?` są czystymi, nieprzechwytującymi grupami, które nie przechwytują tekstu i nie są wliczane do sumy grup.

Wyrażenia regularne

Grupy

- Ważne jest, aby zrozumieć, jak numerowane są grupy, ponieważ niektóre metody klasy `Matcher` przyjmują jako argument liczbę całkowitą określającą konkretny numer grupy:
  - `public int start(int group)`
    Zwraca indeks początkowy podciągu uchwyconego przez podaną grupę podczas poprzedniej operacji dopasowania.
  - `public int end(int group)`
    Zwraca indeks ostatniego znaku, plus jeden, w podciągu przechwyconym przez daną grupę podczas poprzedniej operacji dopasowania.
  - `public String group (int group)`
    Zwraca podciąg wejściowy przechwycony przez daną grupę podczas poprzedniej operacji dopasowywania.

Wyrażenia regularne

Grupy

- Fragment łańcucha wejściowego pasujący do grupy (grup) przechwytujących jest zapisywany w pamięci w celu późniejszego przywołania za pomocą odwołania wstecznego.
- Odwołanie wsteczne jest określane w wyrażeniu regularnym jako odwrotny ukośnik (`\`), po którym następuje cyfra wskazująca numer grupy, która ma zostać przywołana.
- Przykładowo, wyrażenie `(\d\d)` definiuje jedną grupę przechwytującą pasującą do dwóch cyfr z rzędu, którą można przywołać w dalszej części wyrażenia za pomocą odwołania wstecznego `\1`.
- Aby dopasować dowolne dwie cyfry, po których następują dokładnie te same dwie cyfry, jako wyrażenia regularnego należy użyć `(\d\d)\1`.

Wyrażenia regularne

Grupy

- Często używa się grup do wyodrębniania składników dopasowania.
- Przypuśćmy, że mamy element, który opisuje wiersz faktury zawierający nazwę, ilość i cenę jednostkową, taki jak `Blackwell Toaster USD29.95`
- Wyrażenie regularne z grupą dla każdego elementu wygląda tak:

```
(\p{Alnum}+(\s+\p{Alnum}+)*)\s+([A-Z]{3})([0-9.]*)
```

Po dopasowaniu można też wyodrębnić `n`-tą grupę z dopasowania w następujący sposób:

```
String zawartość = matcher.group(n);
```

Wyrażenia regularne

Grupy

- Grupy są numerowane w kolejności występowania ich nawiasów otwierających od 1. Grupa 0 oznacza całość danych wejściowych.
- W poniższym przykładzie widać, w jaki sposób podzielić całe dane wejściowe:

```
Matcher matcher = pattern.matcher(dane);
if (matcher.matches()) {
    nazwa = matcher.group(1);
    waluta = matcher.group(3);
    cena = matcher.group(4);
}
```

- Nie interesuje nas grupa 2, ponieważ powstała poprzez zastosowanie nawiasów użytych do określenia powtórzenia.

Wyrażenia regularne – grupy

- Dla większej przejrzystości można użyć grupy nieprzechwytującej:

```
(\p{Alnum}+(?:\s+\p{Alnum}+)*)\s+([A-Z]{3})([0-9.]*)
```

- Lub, nawet lepiej, przechwytywać za pomocą nazwy:

```
(?<id>\p{Alnum}+(\s+\p{Alnum}+)*)\s+(?<waluta>[A-Z]{3})(?<cena>[0-9.]*)
```

- Wtedy można pobierać elementy za pomocą nazwy:

```
id = matcher.group("id");
```

Wyrażenia regularne

Grupy

- Jeżeli grupa znajduje się wewnątrz powtórzenia w wyrażeniu takim jak w poprzednim przykładzie:

```
(\s+\p{Alnum}+)*
```

  to nie jest możliwe pobranie wszystkich jej dopasowań.
- Metoda `group` zwraca jedynie ostatnie dopasowanie, które rzadko jest przydatne.
- Należy wtedy przechwycić całe wyrażenie za pomocą innej grupy.

Przykłady

- Program `regex/Groups.java`
- Program `regex/HrefMatch.java`

Wyrażenia regularne

Usuwanie lub zastępowanie dopasowań

- Czasem chcemy podzielić w miejscu dopasowania ograniczników i pozostawić wszystko poza dopasowaniem.
- Metoda `Pattern.split` automatyzuje to zadanie. Uzyskujemy tablicę ciągów znaków z usuniętymi ogranicznikami:

```
Pattern przecinki = Pattern.compile("\\s*,\\s*");
String[] tokens = przecinki.split(dane);
// "1, 2, 3" zamienia na ["1", "2", "3"]
```

- Jeżeli istnieje wiele tokenów, można pobrać je w sposób leniwy:

```
Stream<String> tokens =
    przecinki.splitAsStream(input);
```

Usuwanie lub zastępowanie dopasowań

- Jeżeli nie zależy nam na kompilacji wzorca lub leniwym pobieraniu, można po prostu wykorzystać metodę `String.split`:

```
String[] tokens = dane.split("\\s*,\\s*");
```

- Jeżeli chcemy zastąpić wszystkie wystąpienia ciągiem znaków, to należy wywołać metodę `replaceAll` na dopasowaniu:

```
// Normalizuje przecinki
Matcher matcher = przecinki.matcher(input);
String wynik = matcher.replaceAll(", ");
```

Usuwanie lub zastępowanie dopasowań

- Lub, jeśli nie zależy nam na kompilowaniu, to należy wykorzystać metodę `replaceAll` z klasy `String`:

```
String wynik =
    dane.replaceAll("\\s*,\\s*", ",");
```

- Zastępujący ciąg znaków może zawierać numery grup `$n` lub nazwy `$nazwa`. Są one zastępowane zawartością odpowiednich przechwyconych grup:

```
// Zapisuje w wynik "3 godziny i 45 minut"
String wynik = "3:45".replaceAll(
    "(\\d{1,2}):(?<minuty>\\d{2})",
    "$1 godziny i ${minuty} minut");
```

Wyrażenia regularne

Usuwanie lub zastępowanie dopasowań

- Aby użyć znaku `$` lub `\` w zastępującym ciągu znaków, trzeba poprzedzić je znakiem `\`.

Przykład

- Program `regex/RemovingOrReplacingMatches.java`

Wyrażenia regularne

Flagi

- Kilka flag zmienia działanie wyrażeń regularnych. Można je określić podczas kompilowania wzorca:

```
Pattern wzorzec = Pattern.compile(regex,
    Pattern.CASE_INSENSITIVE |
    Pattern.UNICODE_CHARACTER_CLASS
);
```

- Można też określić je wewnątrz wzorca:

```
String regex = "(?iU:wyrażenie)";
```

albo równoważnie:

```
String regex = "(?iU)wyrażenie";
```

Wyrażenia regularne

Flagi

- `Pattern.CASE_INSENSITIVE` lub `i`: dopasowuje znaki niezależnie od wielkości liter. Domyślnie ta flaga bierze pod uwagę jedynie znaki ASCII.
- `Pattern.UNICODE_CASE` lub `u`: użyta w połączeniu z `CASE_INSENSITIVE` wykorzystuje wielkości znaków Unicode przy dopasowywaniu.
- `Pattern.UNICODE_CHARACTER_CLASS` lub `U`: wybiera klasy znaków Unicode zamiast POSIX. Wymusza `UNICODE_CASE`.
- `Pattern.MULTILINE` lub `m`: sprawia, że `^` i `$` są dopasowywane do początku i końca wiersza, a nie do całości danych wejściowych.
- `Pattern.UNIX_LINES` lub `d`: tylko `'\n'` jest znakiem końca linii przy dopasowywaniu `^` i `$` w trybie wielowierszowym.

Wyrażenia regularne

Flagi

- `Pattern.DOTALL` lub `s`: sprawia, że symbol jest dopasowywany do wszystkich znaków, włącznie ze znakami końca linii.
- `Pattern.COMMENTS` lub `x:` białe znaki i komentarze (znaki od # do końca wiersza) są ignorowane.
- `Pattern.LITERAL`: wzorzec jest brany dosłownie i musi być idealnie dopasowany z wyjątkiem możliwości zignorowania wielkości znaków.
- `Pattern.CANON_EQ`: bierze pod uwagę odpowiedniki znaków Unicode. Na przykład znak `u` i następujący po nim znak ¨ (diereza) są dopasowywane do `ü`.

Uwaga: Dwie ostatnie flagi nie mogą być określane wewnątrz wyrażenia regularnego.

Wyrażenia regularne

Flagi wbudowane

Stała	Wyrażenie z flagą wbudowaną
`Pattern.CANON_EQ`	Brak
`Pattern.CASE_INSENSITIVE`	`(?i)`
`Pattern.COMMENTS`	`(?x)`
`Pattern.DOTALL`	`(?s)`
`Pattern.LITERAL`	Brak
`Pattern.MULTILINE`	`(?m)`
`Pattern.UNICODE_CASE`	`(?u)`
`Pattern.UNICODE_CHARACTER_CLASS`	`(?U)`
`Pattern.UNIX_LINES`	`(?d)`

Wyrażenia regularne

Konstrukcje graniczne

Konstrukcja granicza	Opis
^	Początek linii
$	Koniec wiersza
\b	Granica słowa
\B	Granica niebędąca słowem
\A	Początek wejścia
\G	Koniec poprzedniego dopasowania
\Z	Koniec wejścia z wyjątkiem końcowego terminatora, jeśli występuje
\z	Koniec wejścia

Wyrażenia regularne

Przykład (Flagi)

- Program `regex/Flags.java`
- Program `regex/RegexDemo.java`