Power Regular Expressions using Java

Neal Ford

CTO

The DSW Group, Ltd.

www.dswgroup.com

www.nealford.com

nealford.com

The DSW Group, Ltd.

Copyright (c) 2001, The DSW Group, Ltd. All rights reserved.

What This Session Covers:

Regular expressions defined

Using the regex classes in Java

Regular expression techniques

- Patterns
- Groups and subgroups
- Back references
- Greedy, reluctant, and possessive qualifiers
- Lookaheads and lookbehinds

Best practices

Common Regex mistakes

Copyright © 2005, Neal Ford. All rights reserved.

Regular Expressions

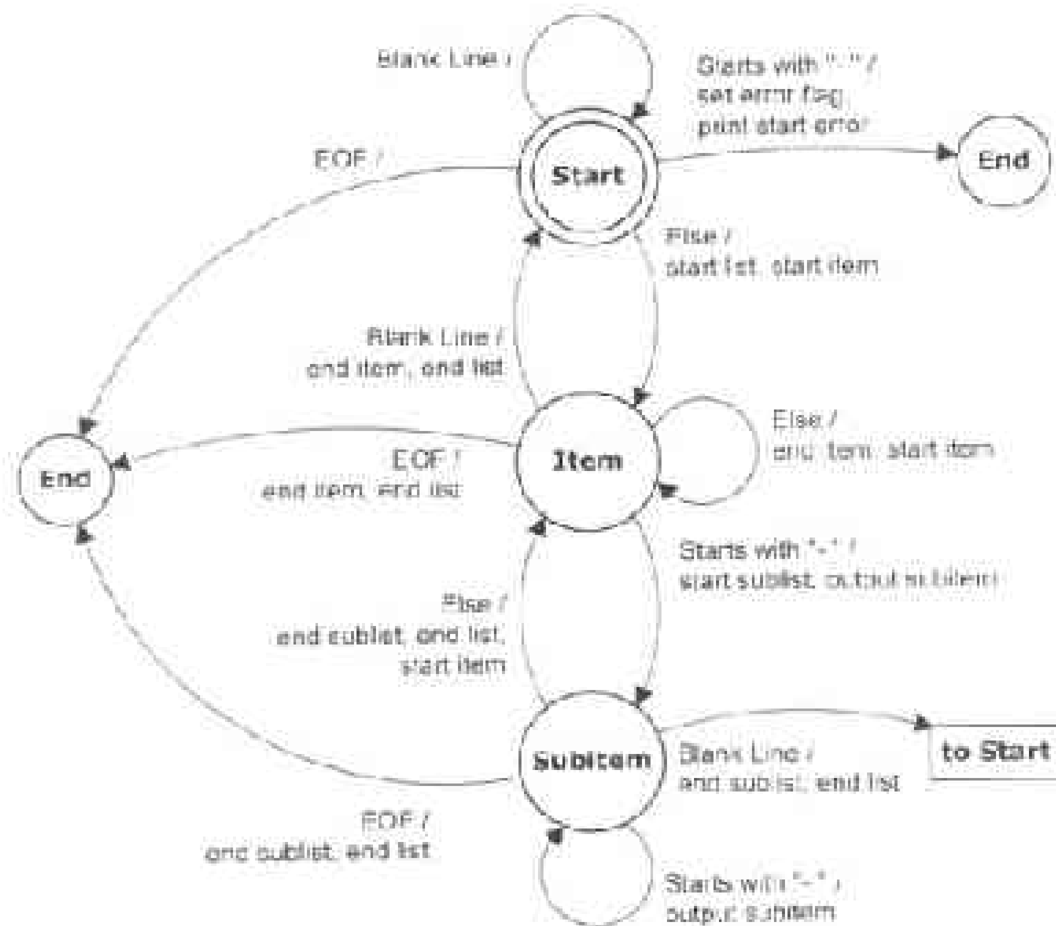
- **Formally defined by information theory as defining the languages accepted by finite automata**
  - Not the typical everyday use
- **Originally developed with neuron sets and switching circuits in mind**
- **Used by compiler writing systems (lex and yacc), text editors, pattern matching, text processing, and logic**

Copyright © 2005, Neal Ford. All rights reserved.

Regex as a FSM

- Regular expressions really define finite state machines

- The Regex matches if you finish the machine in a accepting state



Copyright © 2005, Neal Ford. All rights reserved.

Practical Regular Expressions

- Describe text
- Used for pattern matching in development (editors, command line tools) and programmatically
- Examples:
  - Search and replace
  - grep (Global Regular Expression Print)
    - Thought to come from the ex command G/<regex>/P
  - regex in languages (Perl, Ruby, Java, etc).

Copyright © 2005, Neal Ford. All rights reserved.

Simple Example

- **Let's say you want to verify an email address in the form** `firstname_lastname@somewhere.org` **without regular expressions**
  - Check for an "@" sign
  - Check that the end of the string has ".org" at the end
  - Check for an underscore with letters before and after it
- **This becomes very complex very quickly using String methods**

nealford.com

Copyright © 2005, Neal Ford. All rights reserved.

Simple Example

- Define a regular expression for the string

```
String regex =
     "[A-Za-z]+_[A-Za-z]+@[A-Za-z]+\\.org "
if (email.matches(regex))
   // do something
```

- Regular expressions allow you to exactly and succinctly define matching patterns
- Patterns *describe* text rather than *specifying* it

Copyright © 2005, Neal Ford. All rights reserved.

Regular Expressions in the Wild

Editors

- Emacs/XEmacs
- Eclipse
- JBuilder
- Visual SlickEdit
- IntelliJ

Command line tools

- grep
- find

Not all regular expressions are created equal

Copyright © 2005, Neal Ford. All rights reserved.

Regular Expressions in Java

A combination of several classes

- Pattern
- Matcher
- String class additions
- A new Exception class

Example.

Copyright © 2005, Neal Ford. All rights reserved.

The Pattern Class

Interesting methods

- `static Pattern compile()`
  - Compiles the regex for efficiency
  - Factory class that returns a Pattern object
- `String pattern()`
  - Returns the simple String representing the compiled pattern
- `int flags()`
  - Indicates which flags were used when creating the pattern
- `static boolean matches()`
  - Short-hand way to quickly execute a single match

nealford.com

Copyright © 2005, Neal Ford. All rights reserved.

The Pattern Class

Interesting methods

- `String[] split(CharSequence input)`
  - Similar to StringTokenizer but uses regular expressions to delimit tokens
  - Be careful about your delimiters!
- `String[] split(CharSequence input, int limit)`
  - Limit allows you to control how many elements are returned
  - Limit == 0 returns all matches
  - Limit > 0 returns limit matches
  - Limit < 0 returns as many matches as possible and trailing spaces
    - The value of limit isn't important in this case, just the sign

Copyright © 2005, Neal Ford. All rights reserved.

Regular Expressions: Groups

A *group* is a cluster of characters

Example

- `(\w)(\d\d)(\w+)`
- Defines 4 groups, numbered 0 – 3
  - Group 0: `(\w)(\d\d)(\w+)`
  - Group 1: `(\w)`
  - Group 2: `(\d\d)`
  - Group 3: `(\w+)`
- For candidate string: J50Rocks
  - Group 1: J
  - Group 2: 50
  - Group 3: Rocks

Copyright © 2005, Neal Ford. All rights reserved.

Using Groups

- **Groups allow you to specify operations on strings without knowing the details**
- **From the previous example, you may not know what the string is, but you know the pattern**
  - This allows you to rearrange it without knowing the contents
    - (Group 2)(Group 1)(Group 3)
  - Eclipse example.

Copyright © 2005, Neal Ford. All rights reserved.

The Matcher Class

Interesting Matcher methods

- `Matcher reset()`
  - Clears all state information on the matcher, reverting it to its original state
- `int start()`
  - Returns the starting index of the last successful match
- `int start(int group)`
  - Allows you to specify a subgroup within a match
- `int end()`
  - Returns the ending index of the last successful match + 1
- `int end(int group)`
  - Allows you to specify the subgroup of interest

Copyright © 2005, Neal Ford. All rights reserved.

The Matcher Class

Interesting Matcher methods

- `String group()`
  - Returns the substring of the candidate string that matches the original pattern
- `String group(int group)`
  - Allows you to extract parts of a candidate string that match a subgroup within your pattern
- `int groupCount()`
  - Returns the number of groups the Pattern defines
- `boolean matches()`
  - Returns true the candidate string matches the pattern exactly

Copyright © 2005, Neal Ford. All rights reserved.

The Matcher Class

Interesting Matcher methods

- `boolean find()`
  - Parses just enough of the candidate string to find a match
  - Returns true if a substring is found and parsing stops
  - Returns false if no part of the candidate string matches the pattern
- `boolean find(int start)`
  - Just like its overloaded counterpart except that you can specify where to start searching
- `boolean lookingAt()`
  - Compares as little of the string necessary to achieve a match.

Copyright © 2005, Neal Ford. All rights reserved.

The Matcher Class

String and StringBuffer methods

- ```
  Matcher appendReplacement(
          StringBuffer sb, String replacement
  ```
- ```
  StringBuffer appendTail(
                      StringBuffer sb)
  ```
- `String replaceAll(String replacement)`
- `String replaceFirst(String replacement)`

String class regex methods

- `boolean matches(String regex)`
- ```
  String replaceAll(String regex,
                   String replacement)
  ```
- `boolean split(String regex)`

nealford.com

Copyright © 2005, Neal Ford. All rights reserved.
  ```

Example: Repeat Words

- Using groups and substitutions, you can reference a previous capture within the same regular expression

```
String regex = "\\b(\\w+)(\\1)\\b";
```

- Useful for finding repeated words

Copyright © 2005, Neal Ford. All rights reserved.

Regular Expression Syntax

Pattern	Description
.	Any character
$	End of line
^	Beginning of line
{ }	Ranges
[ ]	Character classes
|	Or
( )	Groups
*	Repeat 0 or more times
+	Repeat 1 or more times
?	Repeat 0 or 1 times

Copyright © 2005, Neal Ford. All rights reserved.

Command & Boundary Characters

Pattern	Description
\d	Any digit [0-9]
\D	A non-digit [^0-9]
\w	Word character [A-Za-z_0-9]
\s	White space
\b	Word boundary

nealford.com
Copyright © 2005, Neal Ford. All rights reserved.

Repeat Characters

Pattern	Repeated
?	0 or 1
*	0 or more
+	1 or more
{n}	Exactly n times
{n,}	At least n times
{n,m}	At least n times but no more than m times. Includes m repetitions

nealford.com
Copyright © 2005, Neal Ford. All rights reserved.

Examples

Phone number.

- `(\d-)?(\d{3}-)?\d{3}-\d{4}`
- `String phoneNum = "(\\d-)?(\\d{3}-)?\\d{3}-\\d{4}"`

Back references

- Allow you to reference groups within the pattern
- In finds: \1, \2, …, \n
  - Look for repeating words: **`\b(\w+)\1\b`**
- In replaces
  - Reorder found groups: $2$3$1

nealford.com

Copyright © 2005, Neal Ford. All rights reserved.

POSIX Character Classes

Pattern	Description
\p{Lower}	A lowercase letter [a-z]
\p{Alpha}	An upper- or lowercase letter
\p{Alnum}	A number or letter
\p{Punct}	Punctuation
\p{Cntrl}	A control character (\x00-\x1F\x7F]
\p{Graph}	Any visible character
\p{Space}	A whitespace character

Copyright © 2005, Neal Ford. All rights reserved.

Regex Game Show Round 1

What does this regex match?

- `[0-9]?[0-9]:[0-9]{2}`
- `[0-9]?[0-9]:[0-9]{2}\s(am|pm)`

General format of time, but with flaws

- Matches 99:99 or 99:99 am
- Better version:
- `(1[012]|[1-9]):[0-5][0-9]\s(am|pm)`

nealford.com
Copyright © 2005, Neal Ford. All rights reserved.

Regex Game Show Round 2

What does this regex match?

- `\d{5}(-\d{4})?`

US Zip Code

Copyright © 2005, Neal Ford. All rights reserved.

Regex Game Show Round 3

What do these regex's match?

- `^(.*)/.*$`
- `^(.*)\\.*$`

The leading path from a filename, from *Nix and Windows

Copyright © 2005, Neal Ford. All rights reserved.

Regex Game Show Round 4

What does this regex match? (Hint: not a standard entity, but a common pattern)

- `^[a-zA-Z]\w{4,15}$`

A password that must

- Start with a character
- Contains only letters, numbers, and underscores
- At least 5 characters
- Maximum of 16 characters

Copyright © 2005, Neal Ford. All rights reserved.

Regex Game Show Round 5

What does this regex match?

- ^#$@.*#$!~%

Not a regular expression: cartoon cursing

Copyright © 2005, Neal Ford. All rights reserved.

Regex Game Show Round 6

- What does this regex match?
  - `\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b`
- Almost an IP Address – what's wrong?
- Pretty good IP address regex (broken up onto multiple lines for spacing)

```
\b(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.
((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){2}
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b
```

Copyright © 2005, Neal Ford. All rights reserved.

Regex Game Show Round 7

What does this regex match?

- ```
  (0[1-9]|1[012])[- /.]
  (0[1-9]|[12][0-9]|3[01])[- /.]
  (19|20)\d\d
  ```

US Date

- In the format mm/dd/yyyy
- The separator may be -, \s, /, or .

nealford.com

Copyright © 2005, Neal Ford. All rights reserved.
  ```

Scrubbing Data

Example: valid US phone numbers

- Handle the digits
  - `(\d{3}-)?\d{3}-\d{4}`
- In Perl, more regex would be added to handle spaces, punctuation, etc.
- In Java, you can easily scrub the data

```
String scrubbed =
    phone.replaceAll("\\p{Punct}|\\s", "");
```

- Now, you can use the simpler expression
  - `(\d{3})?\d{3}\d{4}`

nealford.com

Copyright © 2005, Neal Ford. All rights reserved.

Groups and Subgroups

Groups are groups of characters

Subgroups are smaller groups within the larger whole

Noncapturing subgroups

- Sometimes you want to define a group but you don't want it stored in memory (captured)
- To mark a group as non-capturing, follow the opening parameters with `?:`
  - Example: `(\w)(\d\d)(?:\w+)`
  - Indicates that you won't reference the last group

nealford.com

Copyright © 2005, Neal Ford. All rights reserved.

Greedy Qualifiers

The regex engine tries to match as much as it possibly can

- The pattern `(\w)(\d\d)(\w+)` will match all word characters following the 2 digits

Greedy qualifiers

Pattern	Repeated
?	0 or 1
*	0 or more
+	1 or more
{n}	Exactly n times
{n,}	At least n times
{n,m}	At least n times but no more than m times. Includes m repetitions

Copyright © 2005, Neal Ford. All rights reserved.

Greedy Qualifiers

Given this regex and candidate:

- Candidate: `Copyright 2004`
- Regex: `^.*([0-9]+)`

Match is 4.

Why?

- Greedy ".*" grabs the whole string but has to "give back" digits to match
- Giving back 1 digit is sufficient, and greedy qualifiers are…greedy, so they only give back what they have to

Copyright © 2005, Neal Ford. All rights reserved.

Possessive Qualifiers

Unique to Java!

Greedy and not generous

The regex engine, when encountering (\w+):

- Will try to match as many characters as possible
- Will release those matches if such a release would help a later group achieve a match

Possessive qualifiers prevent this.

- Append a "+" to existing greedy qualifier
  - `(\w++)(\d{2})(\w+)`

Copyright © 2005, Neal Ford. All rights reserved.

Reluctant (Lazy) Qualifiers

- Opposite from greedy qualifiers
- They try to match as little as possible
- Formed by appending “?” to existing greedy qualifiers
  - `X+` => `X+?`
  - `X{n,m}` => `X{n,m}?`
- Controls how the regex engine backtracks
- Example

Copyright © 2005, Neal Ford. All rights reserved.

Lookaheads

Positive lookaheads

- "Peeks" to make sure the pattern exists in the candidate string
- Does not consume the text
- Formed by opening the group with the characters "?=“
  - Example: `(?=\d{2})` confirms that the string has 2 digits in a row

Negative lookaheads

- Allows the engine to confirm that something does not appear in the candidate string
- Formed with "?!"

nealford.com
Copyright © 2005, Neal Ford. All rights reserved.

Regex Game Show Round 8

What is this Regex looking for?

- `,(?=([^']*'[^']*')*(?![^']*'))`

This regex

- Finds a comma
- Looks to make sure that the number of single quotes after the comma is either an even number or 0

Copyright © 2005, Neal Ford. All rights reserved.

Regex Game Show Round 8

```
,(?=([^']*'[^']*')*(?![^']*'))
```

,	Find a comma
(?=	Lookahead to match this pattern:
(	start a new pattern
[^']*'	[not a quote] 0 or many times then a quote
[^'] *'	[not a quote] 0 or many times then a quote, combined with the one above it matches pairs of quotes
)*	end the pattern and match the whole pattern (pairs of quotes) zero, or multiple times
(?!	lookahead to exclude this pattern
[^']*'	[not a quote] 0 or many times then a quote
))	end the pattern

Copyright © 2005, Neal Ford. All rights reserved.

Lookbehinds

Looks to the left in the pattern

Positive lookbehinds

- Confirm the existence of a pattern to the left of the current position
- Formed with “?<=“

Negative lookbehinds

- Confirm the absence of a pattern to the left
- Formed with “?<!”

nealford.com

Copyright © 2005, Neal Ford. All rights reserved.

Using Regular Expressions

- Lots of circumstances pop up where Regex can help
- The *Nix (or Cygwin) find command + grep
  - Find all XML files that are not `web.xml`

```
find . -regex ".*\.xml" | grep -v ".*web.xml"
```

  - Find all XML files that aren't either web or build.xml

```
find . -regex ".*\.xml" | grep -v ".*[web|build].xml"
```

  - Find files (and line numbers) where boundary classes are constructed

```
find . -name "*.java" -exec grep -n  -H
  "new .*Db.*" {} \;
```

nealford.com

Copyright © 2005, Neal Ford. All rights reserved.

Using Regular Expressions

Find all email addresses in all HTML documents in web site

```
find -regex ".*\.html?" -exec grep -n -H
  ".@." {} \; > emails.txt
```

Find all Java source files (except the ones with DB in them) and look for constructor calls

```
find -name "*.java" -not -regex
  ".*Db\.java" -exec grep -H -n
   "new .*Db" {} \;
```

nealford.com

Copyright © 2005, Neal Ford. All rights reserved.

Regex Best Practices

Use noncapturing groups when possible

- Conserves memory use

Precheck your candidate string

- Use String methods to pre-qualify the candidate

Offer the most likely alternative first

- Consider: `*\b(?:MD|PhD|MS|DDS).*`

Be as specific as possible

- Use boundary characters wisely

Copyright © 2005, Neal Ford. All rights reserved.

Regex Best Practices

Specify the position of your match

- `^Homer` is much faster than `Homer`

Specify the size of the match

- If you know the exact number of characters (or a range), use it

Limit the scope of your alternatives

- More efficient
  - To offer small alternatives than large ones
  - Earlier rather than later

nealford.com

Copyright © 2005, Neal Ford. All rights reserved.

Common Regex Mistakes

- Including spaces in the regular expressions (other than spaces you want)
- Not escaping special characters you want treated literally: e.g. '()' instead of '\(\)'
- Forgetting the ^ and $ when you want an entire line to match a regular expression rather than some substring of the line
- Forgetting that something * includes the null string. For example, the regular expression (aaa|bbb)* matches every line!

Copyright © 2005, Neal Ford. All rights reserved.

Questions?

Neal Ford
neal.ford@dswgroup.com
www.nealford.com
www.dswgroup.com

nealford.com

The DSW Group, Ltd.

Copyright (c) 2001, The DSW Group, Ltd. All rights reserved.