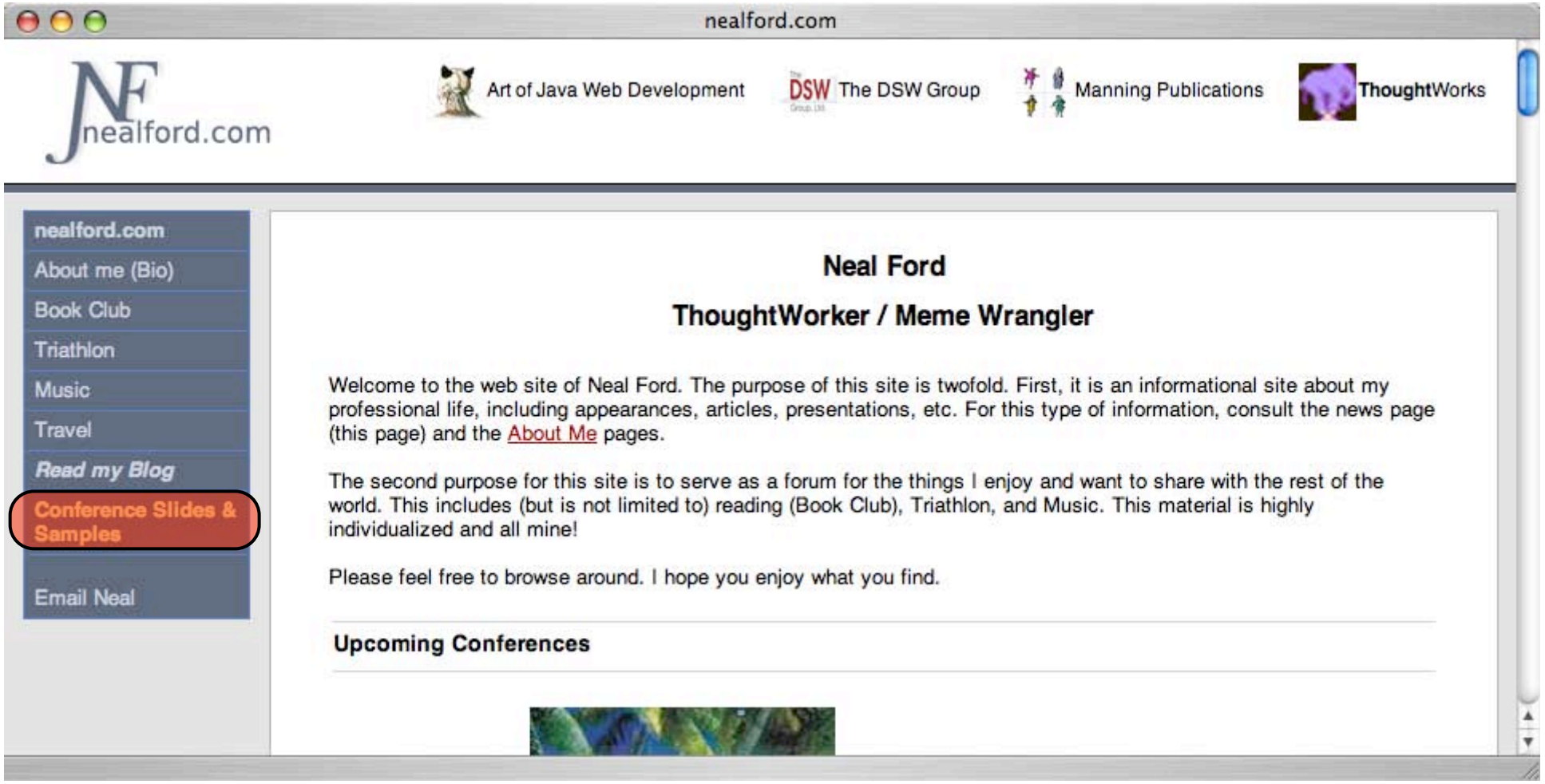
regular expressions in java

NEAL FORD software architect / meme wrangler

ThoughtWorks

nford@thoughtworks.com
3003 Summit Boulevard, Atlanta, GA 30319
www.nealford.com
www.thoughtworks.com
memeagora.blogspot.com


nealford.com
nealford.com
Art of Java Web Development
The DSW Group
Manning Publications
ThoughtWorks
nealford.com
About me (Bio)
Book Club
Triathlon
Music
Travel
Read my Blog
Conference Slides & Samples
Email Neal
Neal Ford
ThoughtWorker / Meme Wrangler
Welcome to the web site of Neal Ford. The purpose of this site is twofold. First, it is an informational site about my professional life, including appearances, articles, presentations, etc. For this type of information, consult the news page (this page) and the About Me pages.
The second purpose for this site is to serve as a forum for the things I enjoy and want to share with the rest of the world. This includes (but is not limited to) reading (Book Club), Triathlon, and Music. This material is highly individualized and all mine!
Please feel free to browse around. I hope you enjoy what you find.
Upcoming Conferences

what i cover

regular expressions defined

using the regex classes in java

lots of regular expression techniques

lots of examples

best practices / pitfalls

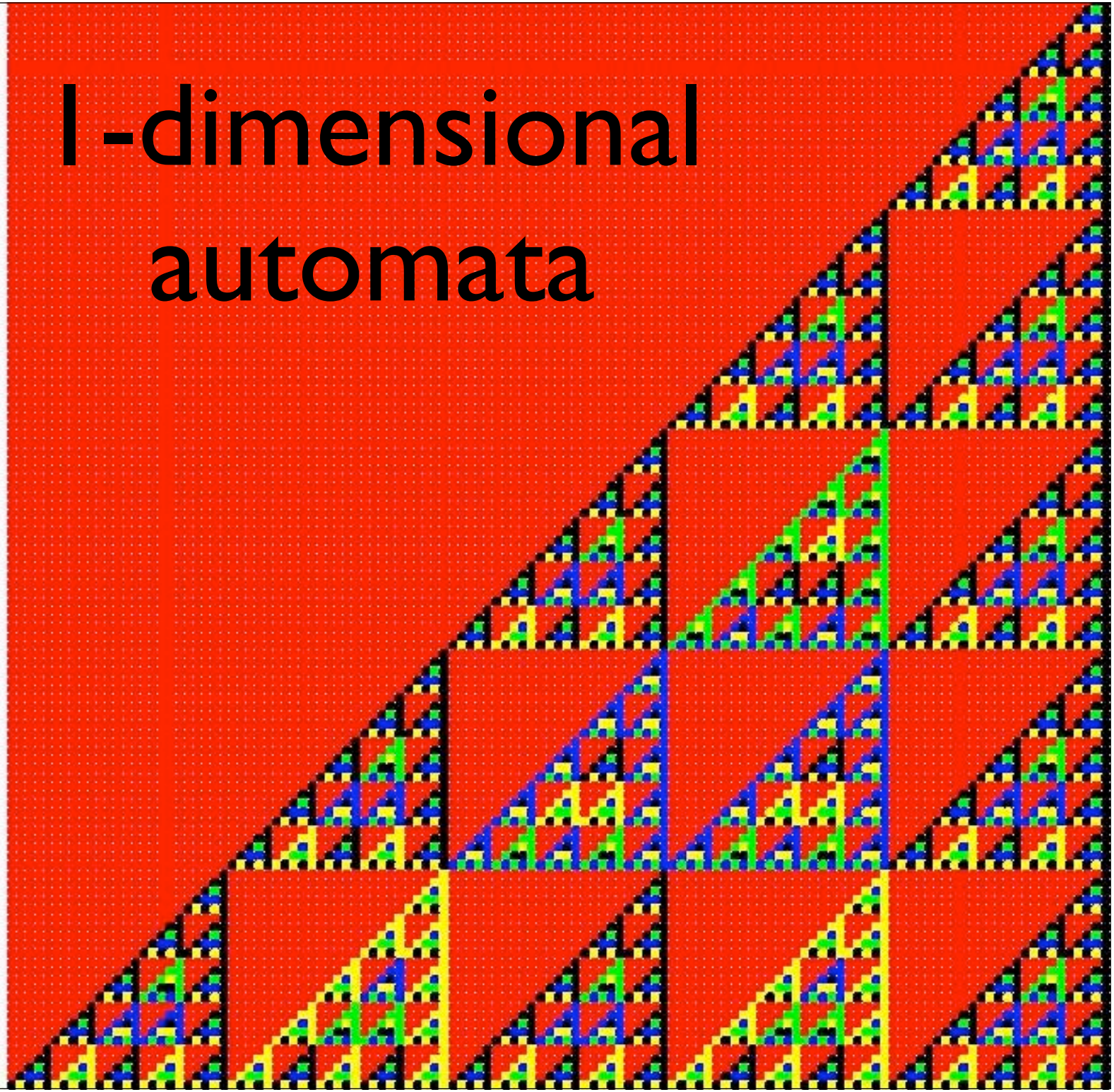
origins

defines the languages accepted by finite automata

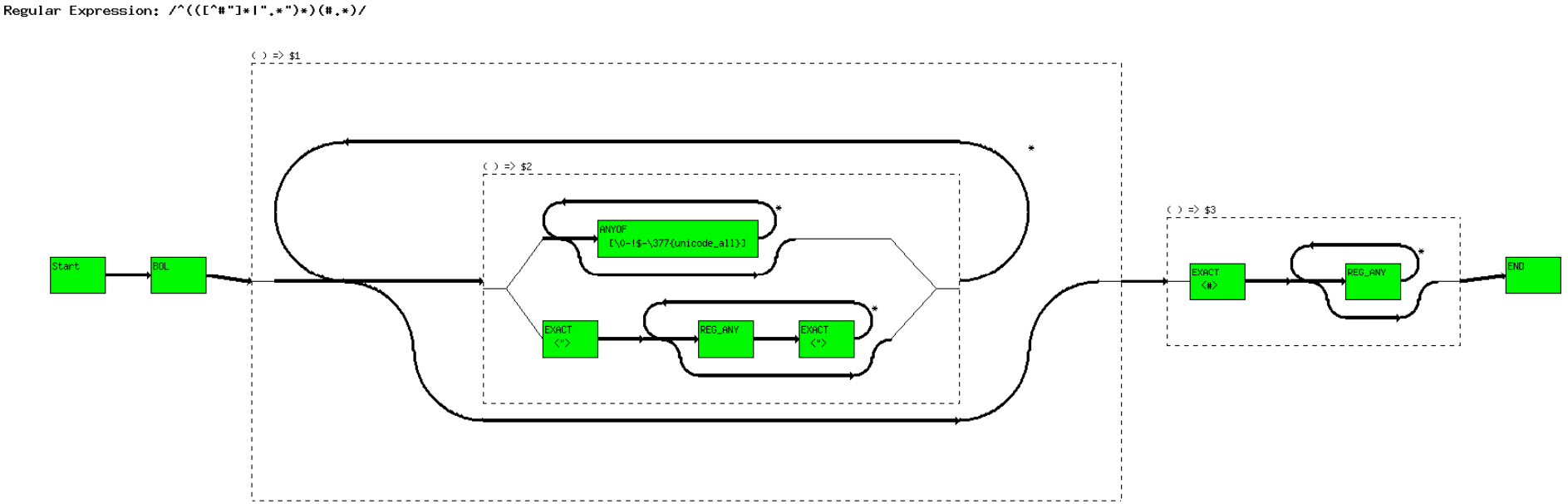
developed for neuron sets & switching circuits

used for a variety of text processing chores

1-dimensional automata



regex as state machines



regex in the wild

editors / ide's

command line tools

g/regular expression/p g/re/p

programming languages


no ansi standard for regex

regular expressions *describe*
text rather than *specify* it

simple example

verify email address in the form
`firstname_lastname@somewhere.org`

check for **`indexOf("@")`**

check **`endsWith(".org")`**

check **`contains("_") &&`**
`indexOf("_") != length &&`
`indexOf("_") != 0`

yuck!

regex version

```
[A-Za-z]+_[A-Za-z]+@[A-Za-z]+\.org
```

```
@Test public void simple_regex() {
    // [A-Za-z]+_[A-Za-z]+@[A-Za-z]+\.org
    String emailRegex =
        "[A-Za-z]+_[A-Za-z]+@[A-Za-z]+\\.org";
    String[] matchers = new String[] {
        "homer_simpson@burns.org",
        "m_burns@burns.org"};
    String[] nonMatchers = new String[] {
        "wsmithers@burns.com",
         "Homer9_simpson@somewhere.org"
    };

    for (String c : matchers)
        assertTrue(c.matches(emailRegex));
    for (String c : nonMatchers)
        assertFalse(c.matches(emailRegex));
}
```

```
public class Candidates {
    private List<String> _good;
    private List<String> _bad;


    public Candidates() {
        _good = new ArrayList<String>(5);
        _bad = new ArrayList<String>(5);
    }

    public List<String> thatMatch() {
        return _good;
    }

    public List<String> notMatching() {
        return _bad;
    }

    public Candidates addMatchers(String goodCandidate) {
        _good.add(goodCandidate);
        return this;
    }

    public Candidates addNonMatchers(String badCandidate) {
        _bad.add(badCandidate);
        return this;
    }
}
```

```
@Test public void Simple_regex() {
    // [A-Za-z]+_[A-Za-z]+@[A-Za-z]+\.org
    String emailRegex =
        "[A-Za-z]+_[A-Za-z]+@[A-Za-z]+\\.org";

    candidates.addMatchers("homer_simpson@burns.org").
               addMatchers("m_burns@burns.org");

    candidates.addNonMatchers("wsmithers@burns.com").
               addNonMatchers("Homer9_simpson@somewhere.org");

    for (String c : candidates.thatMatch())
        assertTrue(c.matches(emailRegex));
    for (String c : candidates.notMatching())
        assertFalse(c.matches(emailRegex));
}
```

regular expressions allow you to precisely & succinctly match text

escaping

escape “special characters” in regular expressions with “\”

same as in java

“crufts up” java strings with regex

```
String regex =
    "[A-Za-z]+_[A-Za-z]+@[A-Za-z]+\\.org"

     [A-Za-z]+_[A-Za-z]+@[A-Za-z]+\.org
```

regex support in java

added in sdk 1.4.x

minor changes in 1.5.x

3 new classes: `Pattern, Matcher, Exception`

modified classes: `String`, `StringBuffer`

added `CharSequence`

Pattern

```
static Pattern compile(s)
static Pattern compile(s, flags)
```

compiles regex for efficiency

factory that returns a **`Pattern`** instance

```
String pattern()
```

```
int flags()
```

Pattern

`static boolean matches()`

execute a one-off match

used by other convenience **`matches()`** methods

`String split[](CharSequence input)`

`StringTokenizer` on steroids

```
@Test public void Simple_regex_with_pattern() {
    // [A-Za-z]+_[A-Za-z]+@[A-Za-z]+\.org
    String emailRegexSample =
        "[A-Za-z]+_[A-Za-z]+@[A-Za-z]+\\.org";
    candidates.addMatchers("homer_simpson@burns.org").
               addMatchers("m_burns@burns.org");

    candidates.addNonMatchers("wsmithers@burns.com").
               addNonMatchers("Homer9_simpson@somewhere.org");

    Pattern p = Pattern.compile(emailRegexSample);
    Matcher m;
    for (String c : candidates.thatMatch()) {
        m = p.matcher(c);
        assertTrue(m.matches());
    }
    for (String c : candidates.notMatching())
        assertFalse(p.matcher(c).matches());
}
```

```java
@Test public void Case_insensitive_searchs_with_pattern_flags() {
    String regex = "A really long sentence";
    String candidate = "A really long SENTENCE";
    Pattern p = Pattern.compile(regex, Pattern.CASE_INSENSITIVE);
    Matcher m = p.matcher(candidate);
    assertTrue(m.matches());
}
```

groups

clustering of characters

```
(\w)(\d\d)(\w*)
```

defined in the regex but *realized* in the match

	J50Rocks
(\w)(\d\d)(\w*)	J50Rocks
(\w)	J
(\d\d)	50
(\w*)	Rocks

Matcher

`boolean matches()`

regex matches candidate completely

`boolean find()`

parses just enough of candidate to match

returns true until no more matches

```
@Test public void Odyssey_search_with_find() {
    String regexToMatch = "Athene";
    Pattern p = Pattern.compile(regexToMatch);
    Matcher m = p.matcher(odysseyPartOne);
    int count = 0;
    while (m.find()) {
        count++;
        assertThat(m.group(), is("Athene"));
    }
    assertThat(count, is(5));
}
```

Matcher

`Matcher reset()`

clears existing matcher state

`int start(), int start(int group)`

starting index of last successful match

Matcher

`int end(), int end(int group)`

ending index of last successful match + 1

`String group(), String group(int groupNo)`

contents of matched group

`int groupCount()`

```
@Test public void groups() {
    // (\w)(\d)
    String regex = "(\\w)(\\d)";
    String candidate = "J2 comes before J5, which both come before H7";
    String[] matchers = { "J2", "J5", "H7"};
    Matcher m = Pattern.compile(regex).matcher(candidate);
    int i = 0;
    while (m.find()) {
        assertThat(m.group(), is(matchers[i]));
        assertThat(m.group(1), is(matchers[i].substring(0, 1)));
        assertThat(m.group(2), is(matchers[i].substring(1, 2)));
        i++;
    }
}
```

back references

reference groups previously defined within the regex

```
\b(\w+)\1\b
```

finds words with repeating parts:

“froofroo” “tutu”

```
@Test public void repeat_words() {
    //  \b(\w+)\1\b
    String regex = "\\b(\\w+)\\1\\b";
    String candidate =
        "The froofroo tutu cost more than than the gogo boots.";
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(candidate);
    String[] matchers = { "froo", "tu", "go" };
    int i = 0;
    while (m.find())
        assertThat(m.group(1), is(matchers[i++]));
}
```

syntax

Pattern	***Description***
.	Any character
^	Beginning of line
$	End of line
[]	Character classes*
	[A-Za-z] [0-9] [.]
|	or operator
( )	Group delimiter

command & boundary

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

multiplicity

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

numbered repetitions

phone numbers & death by \

```
^\((\d{3})\)\s(\d{3})-(\d{4})$

"^\\((\\d{3})\\)\\s(\\d{3})-(\\d{4})$"
```

alternative: [(]

```
"^[(](\\d{3})[)]\\s(\\d{3})-(\\d{4})$"
```

```java
@Test public void phone_number() {
    //                  ^\((\d{3})\)\s(\d{3})-(\d{4})$
    String regex = "^\\((\\d{3})\\)\\s(\\d{3})-(\\d{4})$";
    String phoneNumber = "(404) 555-1234";
    Pattern pattern = Pattern.compile(regex);
    Matcher m = pattern.matcher(phoneNumber);
    assertTrue(m.matches());
    assertThat(m.group(1), is("404"));
    assertThat(m.group(2), is("555"));
    assertThat(m.group(3), is("1234"));
}
```

```java
@Test public void phone_number_alternative_to_slash_death() {
    //                ^[(](\d{3})[)]\s(\d{3})-(\d{4})$
    String regex = "^[(](\\d{3})[)]\\s(\\d{3})-(\\d{4})$";
    String phoneNumber = "(404) 555-1234";
    Pattern pattern = Pattern.compile(regex);
    Matcher m = pattern.matcher(phoneNumber);
    assertTrue(m.matches());
    assertThat(m.group(1), is("404"));
    assertThat(m.group(2), is("555"));
    assertThat(m.group(3), is("1234"));
}
```

triple uses for groups

groups of characters within the regex match

multiplicity operators for groups of characters

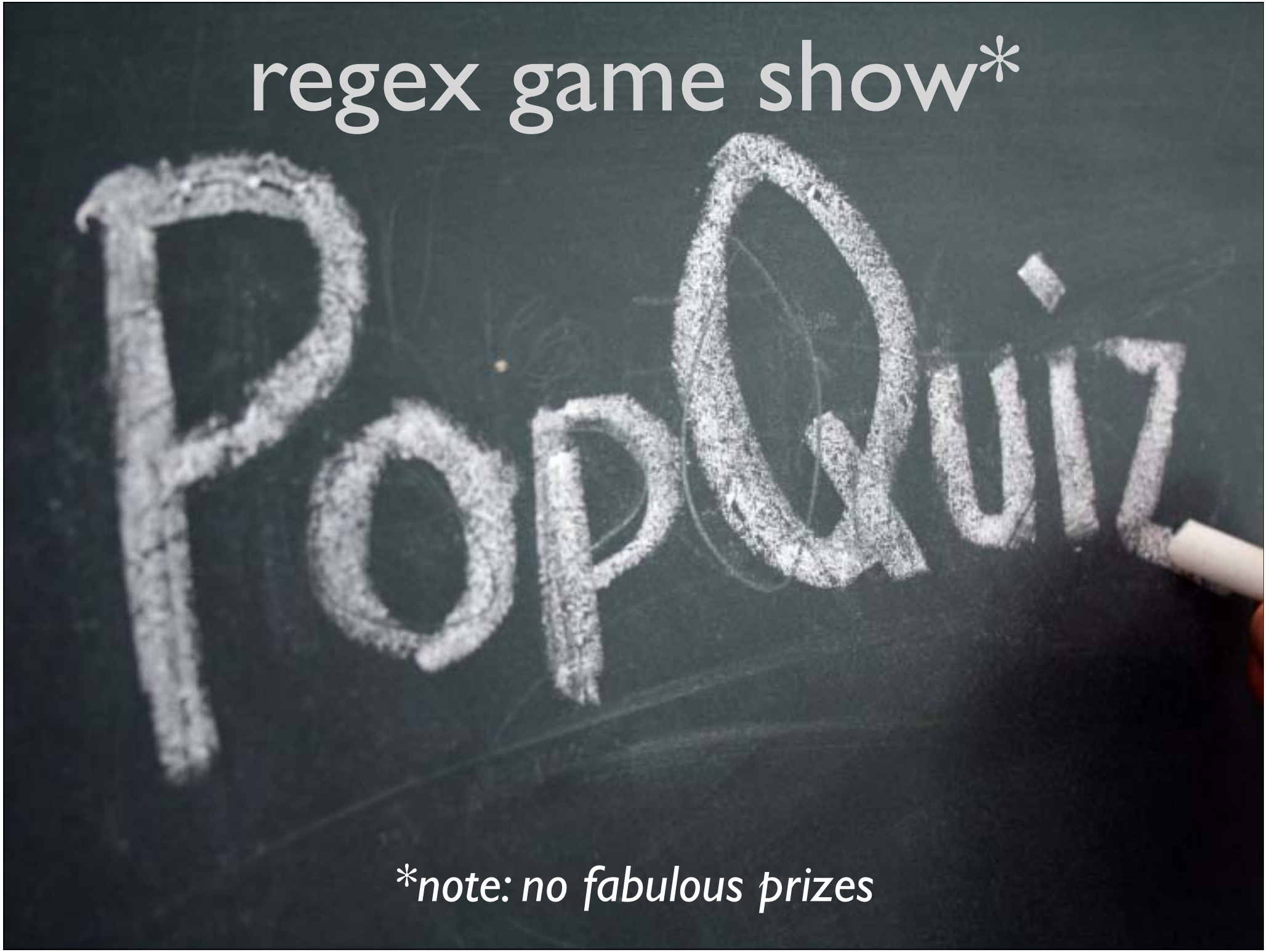
```
(\d{3}-)?\d{3}-\d{4}
```

special flagged behaviors

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

regex game show*



*note: no fabulous prizes

round 1

```
[0-9]?[0-9]:[0-9]{2}
```

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

time

```
(1[012]|[1-9]):[0-5][0-9]\s(am|pm)
```

round 2

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

us zip code (with optional +4)

round 3

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

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

leading part of a filename

```
/usr/bin/local/foo

c:\Program Files\JetBrains\IntelliJ\IDEA.exe
```

round 4

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

hint: something you see all the time

password that:

- ✓ start with letter
- ✓ only letters, numbers, & underscores
- ✓ minimum 5, maximum 16

round 5

cartoon cursing

round 6

```
\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b
```

ip address ?

```
\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
```

round 7

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

us date `- /.`

```
(0[1-9]|1[012])([- /.])
(0[1-9]|[12][0-9]|3[01])\2
(19|20)\d\d
```

round 8

```
\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b
```

pretty good email address

“official” email address

```
?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"
(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\
[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*
[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:25[0-5]|
2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|
2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:
[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\
[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

round 9

```
(BB)|([^B]{2})
```

shakespearean regex

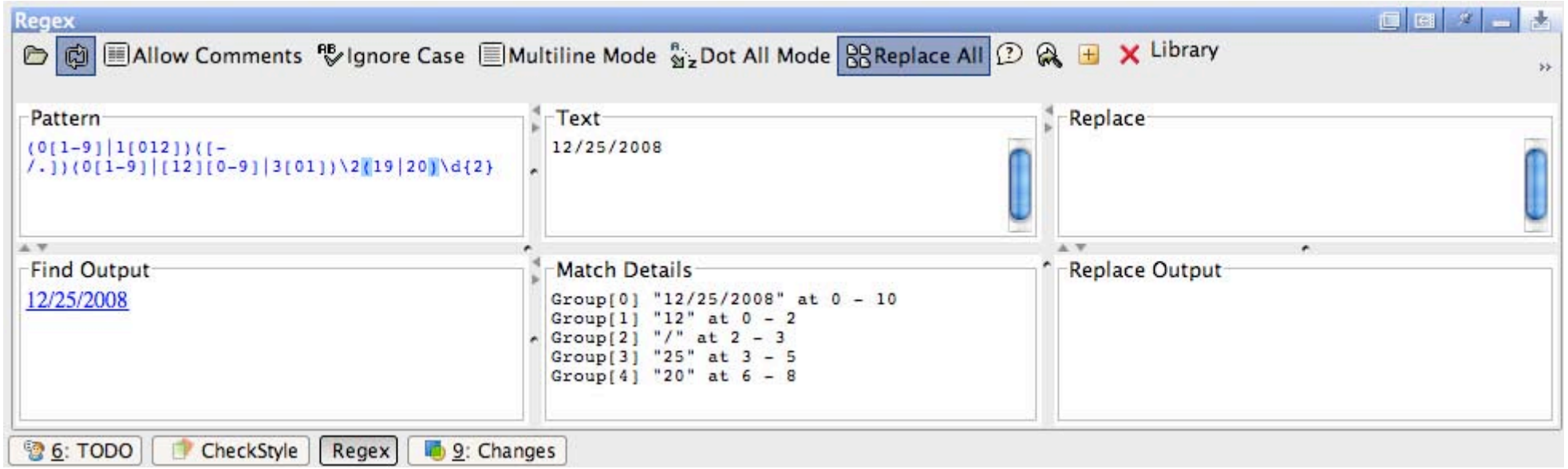
“To be or not to be”
“BB or not BB”
“Call me Ishmael”

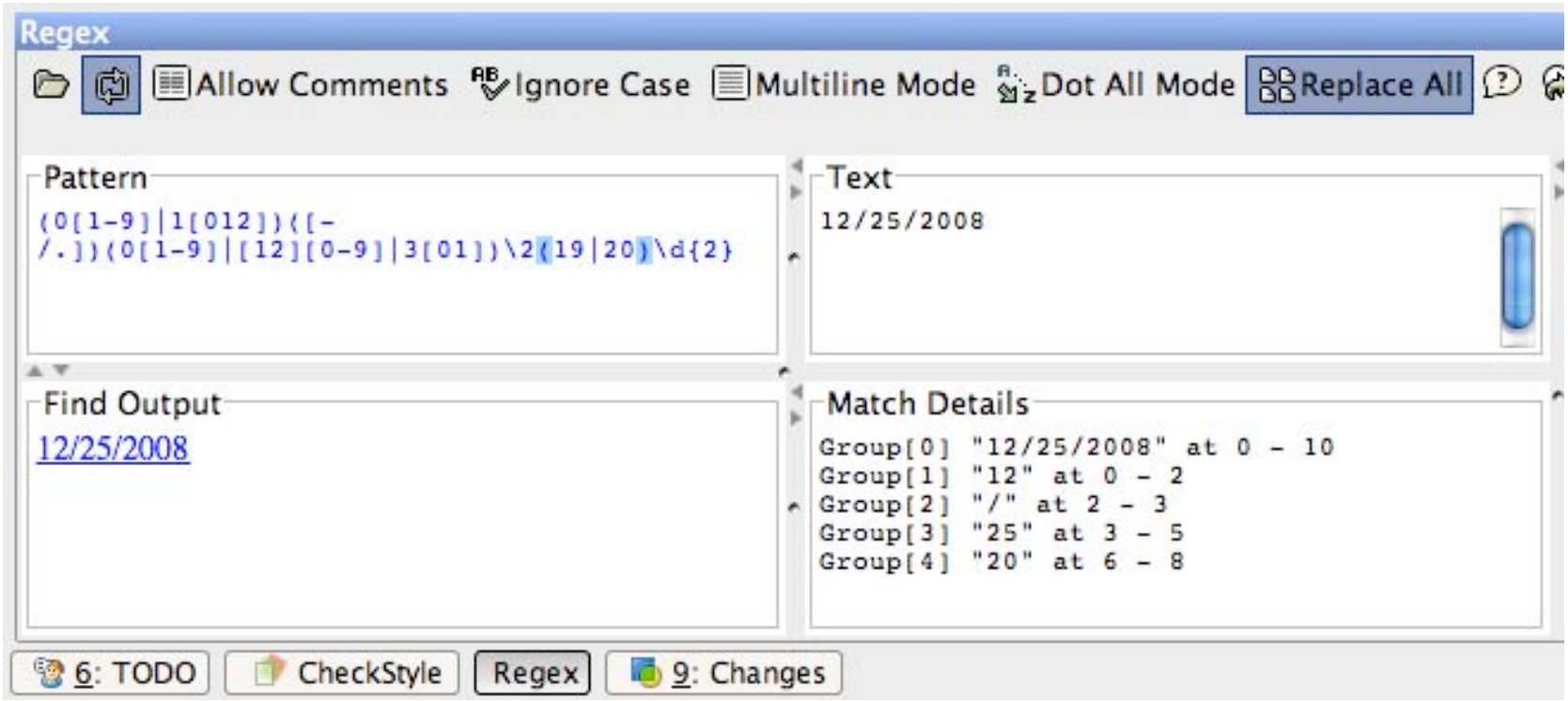
tool support

both intellij & eclipse have regex plug-ins

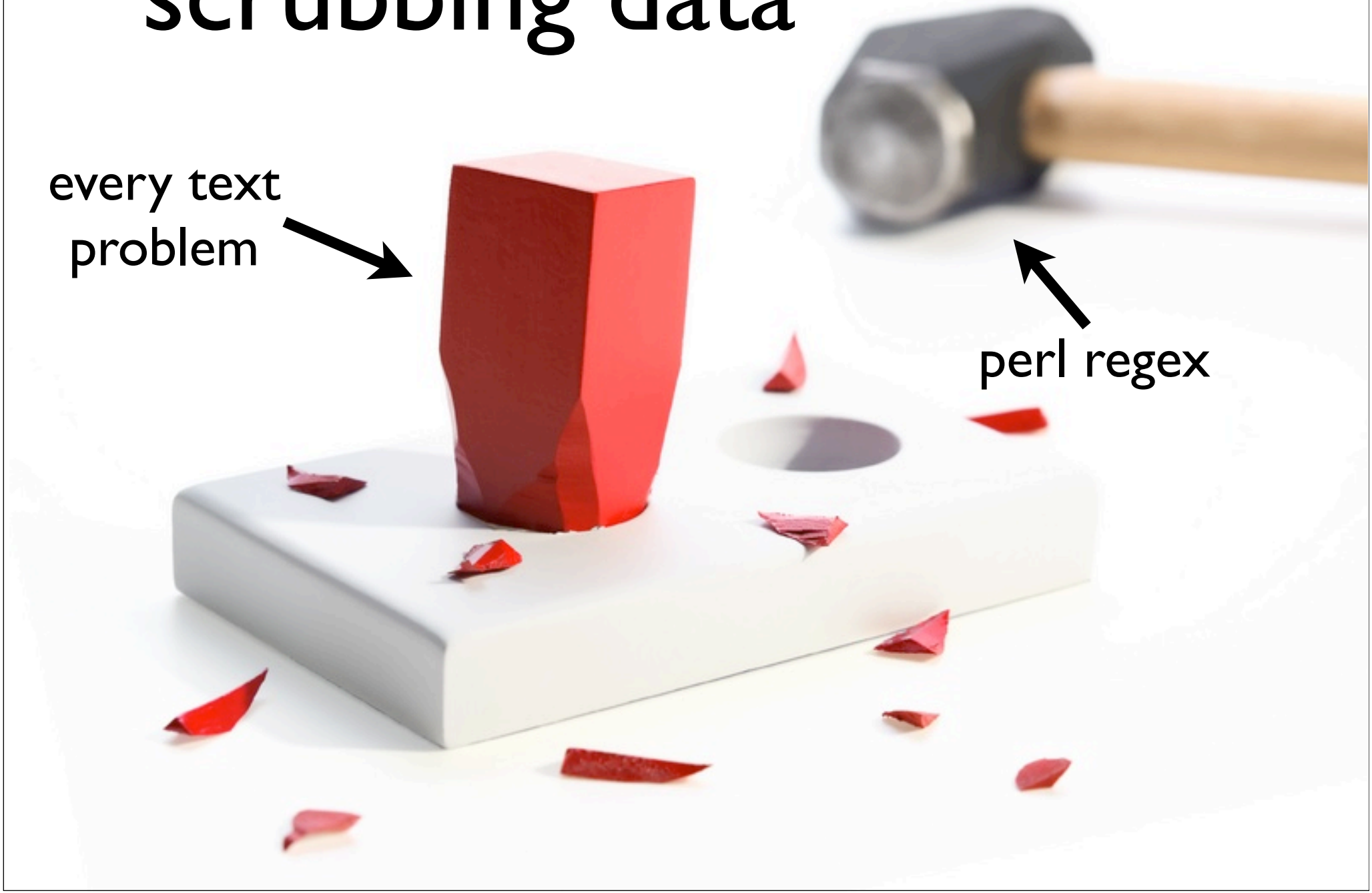
expresso (for .net)

numerous on-line regex checkers


Regex
Allow Comments
Ignore Case
Multiline Mode
Dot All Mode
Replace All
Library
Pattern
(0[1-9]|1[012])([-/.])(0[1-9]|[12][0-9]|3[01])\2(19|20)\d{2}
Text
12/25/2008
Replace
Find Output
12/25/2008
Match Details
Group[0] "12/25/2008" at 0 - 10
Group[1] "12" at 0 - 2
Group[2] "/" at 2 - 3
Group[3] "25" at 3 - 5
Group[4] "20" at 6 - 8
Replace Output
6: TODO
CheckStyle
Regex
9: Changes


Regex
Allow Comments
Ignore Case
Multiline Mode
Dot All Mode
Replace All
Pattern
(0[1-9]|1[012])([-/.])(0[1-9]|[12][0-9]|3[01])\2(19|20)\d{2}
Text
12/25/2008
Find Output
12/25/2008
Match Details
Group[0] "12/25/2008" at 0 - 10
Group[1] "12" at 0 - 2
Group[2] "/" at 2 - 3
Group[3] "25" at 3 - 5
Group[4] "20" at 6 - 8
6: TODO
CheckStyle
Regex
9: Changes

scrubbing data



scrubbing data

```
@Test public void phone_numbers() {
    // (\d{3})?\d{3}\d{4}
    String regex = "(\\d{3})?\\d{3}\\d{4}";
    candidates.
            addMatchers("404-555-1234").
            addMatchers("(404) 555  - 1234").
            addMatchers("404.555.1234");
    for (String num : candidates.thatMatch()) {
        String scrubbed = num.replaceAll("\\p{Punct}|\\s", "");
        assertTrue(Pattern.compile(regex).
                matcher(scrubbed).matches());
    }
}
```

```
@Test public void scrub_ip_address() {
    String regex = "^(\\d{1,3})[.](\\d{1,3})[.](\\d{1,3})[.](\\d{1,3})$";
    candidates.addMatchers("255.255.255.255")
              .addMatchers("0.0.0.0")
              .addMatchers("125.10.0.245");
    candidates.addNonMatchers("256.0.0.0");
    Pattern p = Pattern.compile(regex);
    for (String ip : candidates.thatMatch()) {
        Matcher m = p.matcher(ip);
        for (int i = 1; i <= 4; i++) {
            assertTrue(m.matches());
            int triad = Integer.parseInt(m.group(i));
            assertTrue(triad >= 0 && triad <= 255);
        }
    }
    for (String ip : candidates.notMatching()){
        Matcher m = p.matcher(ip);
        boolean badNum = false;
        for (int i = 1; i <= 4; i++) {
            assertTrue(m.matches());
            int triad = Integer.parseInt(m.group(i));
            if (triad < 0 || triad > 255)
                badNum = true;
        }
        assertTrue(badNum);
    }
}
```

scrubbing ip's


group modifiers

non-capturing

define a group you don't want to reference in the match

primarily used to apply multiplicity without capturing the contents

add the sigil `?:` to the beginning of the group

```
(?:\d{4}-)?\d{3}-\d{4}
```

does not appear in **`groupCount()`**

```
@Test public void non_capturing_groups() {
    // (?:\d{4}-)?\d{3}-\d{4}
    String regex = "(?:\\d{3}-)?\\d{3}-\\d{4}";
    String candidate = "404-555-1234";
    Matcher m = Pattern.compile(regex).matcher(candidate);
    assertTrue(m.matches());
    assertThat(m.groupCount(), is(0));
}
```

```
@Test public void more_non_capturing_sub_groups() {
    // (?:<li>)(.*)(?:</li>)
    String regex = "(?:<li>)(.*)(?:</li>)";
    String candidate = "<li>Now is the time for all good men</li>";
    Matcher m = Pattern.compile(regex).matcher(candidate);
    assertTrue(m.matches());
    assertThat(m.groupCount(), is(1));
    assertThat(m.group(),
            is("<li>Now is the time for all good men</li>"));
    assertThat(m.group(1),
            is("Now is the time for all good men"));
}
```

some of the 7 deadly sins & regex

greediness

Pattern	*Description*
+	Match 1 or more
*	Match 0 or more

modify the thing to their left

try to match *as many characters as possible*

consider:

```
^.*([0-9]+)
```

```
copyright 2004
```

what is the group 1 match?

4

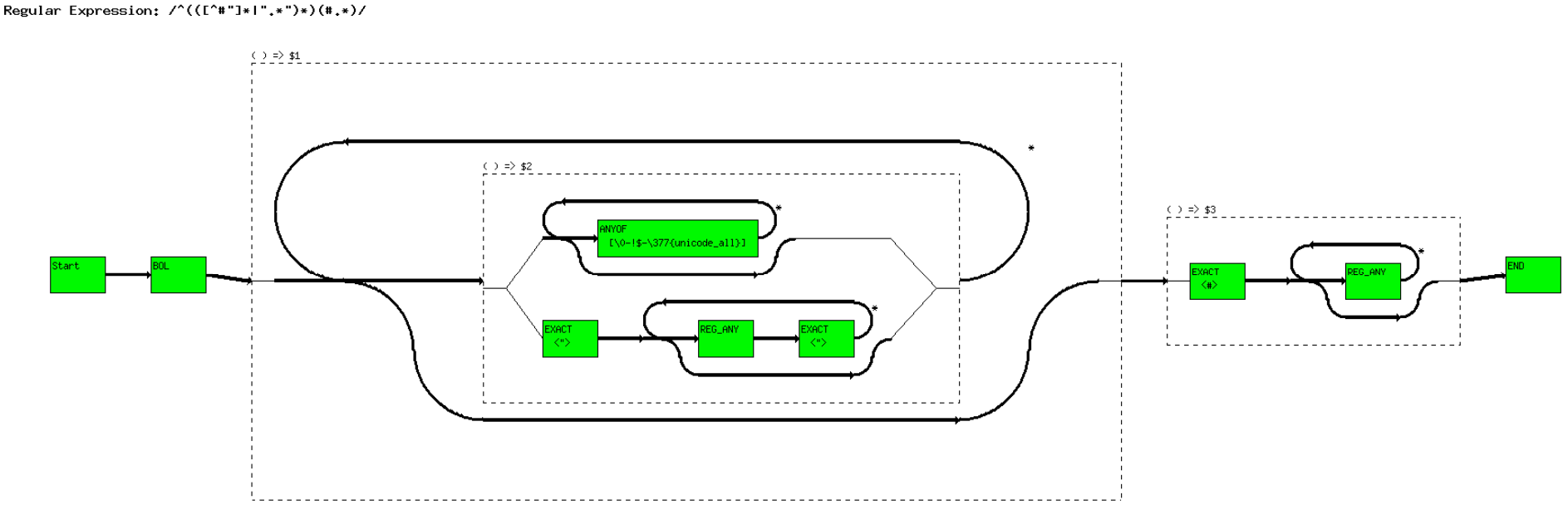
```
copyright 2004
```

“greedy but generous”

the name "regex" abbreviates "Regular Expression"

```
@Test public void greedy_backtrack() {
    // ".*"
    String regex = "\".*\"";
    String candidate =
        "The name \"regex\" abbreviates \"Regular Expression\".";
    Matcher m = Pattern.compile(regex).matcher(candidate);
    assertTrue(m.find());
    assertThat(m.group(),
            is("\"regex\" abbreviates \"Regular Expression\""));
}
```

deterministic →



non-deterministic ↔

possessive

greedy but not generous

add + to existing greedy operator

```
(\w++)(\d{2})(\w+)
```

```java
@Test public void possessive() {
    String greedyRegex = "(\\w+)(\\d{2})(\\w+)";
    String candidate = "Java50Rocks";
    String possessiveRegex = "(\\w++)(\\d{2})(\\w+)";
    assertFalse(candidate.matches(possessiveRegex));
    Matcher m = Pattern.compile(greedyRegex).matcher(candidate);
    assertTrue(m.matches());
    assertThat(m.group(1), is("Java"));
    assertThat(m.group(2), is("50"));
    assertThat(m.group(3), is("Rocks"));
}
```

reluctant (lazy)

“overly generous”

try to match as little as possible

add ? to existing greedy qualifier

```
^.*?([0-9]+)
```

```java
@Test public void reluctance() {
    // ^.*?([0-9]+)
    String regex = "^.*?([0-9]+)";
    String candidate = "Copyright 2004";
    Matcher m = Pattern.compile(regex).matcher(candidate);
    assertTrue(m.find());
    assertThat(m.group(1), is("2004"));
    assertFalse(m.find());
}
```

copyright 2004

```java
@Test public void lazy_backtrack() {
    // ".*?"
    String regex = "\".*?\"";
    String candidate =
            "The name \"regex\" abbreviates \"Regular Expression\".";
    Matcher m = Pattern.compile(regex).matcher(candidate);
    assertTrue(m.find());
    assertThat(m.group(), is("\"regex\""));
    assertTrue(m.find());
    assertThat(m.group(), is("\"Regular Expression\""));
    assertFalse(m.find()) ;
}
```

positive lookaheads

"peeks" at the upcoming parts of the candidate

does not consume text

add ?= as group modifier

```java
@Test public void simple_lookahead() {
    // <li>(.*)(?=</li>)
    String regex = "<li>(.*)(?=</li>)";
    String candidateThatMatches = "<li>list item</li>";
    String nonMatchingCandidate = "<li>list item";

    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(candidateThatMatches);
    assertTrue(m.find());
    assertThat(m.group(1), is("list item"));
    assertFalse(p.matcher(nonMatchingCandidate).find());
}
```

negative lookaheads

"peeks" to ensure the string does not appear

does not consume text

add `?!` as group modifier

```
@Test public void file_name_with_negative_lookahead() {
    // .*[.](?!bat$|exe$).*$
    String regex = ".*[.](?!bat$|exe$).*$";
    candidates.
            addMatchers("foo.batch").
            addMatchers("foo.doc").

            addNonMatchers("foo.bat").
            addNonMatchers("foo.exe");
    Pattern p = Pattern.compile(regex);
    for (String s : candidates.thatMatch())
        assertTrue(p.matcher(s).matches());
    for (String s : candidates.notMatching())
        assertFalse(p.matcher(s).matches());
}
```

look behinds

look at the left of the current position

positive look behinds verify text is present

add ?<= as group modifier

negative look behinds verify the absence of text

add ?<! as group modifier

```
@Test public void wrap_around() {
    // (?<=<li>)(.*)(?=</li>)
    String regex = "(?<=<li>).*(?=</li>)";
    String candidate = "<li>Now is the time for all good men</li>";
    Matcher m = Pattern.compile(regex).matcher(candidate);
    assertTrue(m.find());
    assertThat(m.groupCount(), is(0));
    assertThat(m.group(), is("Now is the time for all good men"));
}
```

look “x” modifiers do not consume text

round 10

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

hint: finds a comma

verifies that the number of single quotes after the comma is either 0 or an even number

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

Pattern	Description
`,`	Find a comma
`(?=`	Lookahead pattern to match...
`(`	Start a new pattern
`[^']*'`	[not a quote] 0 or many times, then a quote
`[^']*'`	[not a quote] 0 or many times, then a quote
`)*`	End the patterns; match 0 or more times
`(?!`	Lookahead to exclude...
`[^']*'`	[not a quote] 0 or many times, then a quote
`))`	End both patterns

cli regex

both *-nix & cygwin have great regex support

all xml files except build.xml or web.xml

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

all java sources that have this pattern

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

cli regex

all java sources (except ones with DB in the name) with constructor calls

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

all public & private methods with **`ShoppingCart`** parameter

```
find -name *.java -exec egrep -n -H
      "(public|private).*\(ShoppingCart\b.*" {} \;
```

best practices

use non-capturing groups when you don't need to harvest results

pre-check the candidate using **`String`** methods

offer the most likely alternatives first

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

use boundary characters wisely

best practices

specify the position of the match

`^Homer` is faster than `Homer`

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

limit the scope of alternatives

smaller alternatives earlier

common mistakes

including spaces

not escaping special characters

forgetting ^ and $

x* includes the null string

(aaa|bbb)* matches everything

questions?

please fill out the session evaluations
slides & samples available at nealford.com

NEAL FORD software architect / meme wrangler

ThoughtWorks

nford@thoughtworks.com
3003 Summit Boulevard, Atlanta, GA 30319
www.nealford.com
www.thoughtworks.com
memeagora.blogspot.com

resources

Mastering Regular Expressions
Jeffrey E. F. Friedl