CS416
Introduction to Computer Science II
Spring 2018

09 Regular Expressions

Previously in 416

- Data structures
  - Organizing collections of related data objects
  - Linear organizations: arrays, lists, stacks, queues
  - Nonlinear: hash tables, trees

Preview

- String processing
  - defining string patterns
  - matching patterns
  - pattern substitution
- String processing applications

Preview

- String processing
  - defining string patterns
  - matching patterns
  - pattern substitution
- String processing applications

Regular Expressions

- A *regular expression* defines a string *pattern*
  - a pattern can be used to define a valid Java identifier: *an unlimited-length sequence of letters, digits, the dollar sign "$", or the underscore character "_"; the sequence must start with a letter*.
- Need a notation (actually it's a simple language) to define patterns such as this.
- Need a notation/interface for using the patterns

Regular Expression Processor

- A *regular expression processor*:
  - takes as input a *string* and a *pattern*
  - decides if/where the pattern is found in the string
  - can also <u>replace</u> the piece of the string that matches the pattern with another string
    - patterns can have multiple parts, which can be used as part of the replacement string
    - Example: convert a list of names from "Last, First" to "First Last"

String: String to be searched
Pattern: to be

String: String <u>**to be**</u> searched

Replace: being
String: String <u>**being**</u> searched

Pattern: (Last), (First)
Part 1: Last
Part 2: First
Replace: First Last

The Pattern Class

- Java's *Pattern* class parses regular expressions
- Simplest invocation is a *static* method:
  - *boolean Pattern.matches (String regex, String test);*

```
// check if address is in Durham or Dover
String town = getNextTown(); // user method

if (  Pattern.matches( "Durham", town )
   || Pattern.matches( "Dover", town ))
{
  System.out.println( "Found: " + town );
}
```

Requires that the pattern match the *entire* string.

```
// the "or" can be in regex: more efficient
if ( Pattern.matches( "Durham|Dover", town )
. . .
```

String also has a *matches* method:

```
if ( town.matches( "Durham" ))
```

“Wildcard” matches

- Suppose we don't have a separate *town* field, but just have a string containing the entire address.
  - We want to see if “Durham” appears <u>anywhere</u> in the address field
  - `.*` in a regex matches *0 or more* of *<u>any</u>* character

```
// check if address is in Durham or Dover
String addr = getNextAddress();

if (  addr.matches( ".*Durham.*" ) )
{
  System.out.println( "Found: " + addr );
}
```

Pattern matches any number of any characters, followed by "Durham", followed by any number of any characters.

```
// can also write:
if ( addr.matches( ".*Durham.*|.*Dover.*" ) )
```

How does it work?

- To understand how to use regular expressions well, need to understand the basic matching model, especially for pattern *replications* (wildcards).
- Demonstrate by over-simplified example. There are 2 key concepts:
  - replication matching is greedy; it tries to match as much as possible
  - subsequent match failure causes algorithm to backup and try a less greedy match, if possible, and then go forward again.

Step	*Comment*	*Pattern*	*Input String*	*Repl#*
1	Match .* greedily	.*Durham.*	1 Main St. Durham, NH	21
2	Match *Durham* fails; match .* less greedily	.*Durham.*	1 Main St. Durham, NH	20
3	Match *Durham* fails; match .* less greedily	.*Durham.*	1 Main St. Durham, NH	19
. . .	18, 17, ... , 12	.*Durham.*	1 Main St. Durham, NH	12
11	Match *Durham* fails; match .* less greedily	.*Durham.*	1 Main St. Durham, NH	11
12	Match *Durham* succeeds	.*Durham.*	1 Main St. Durham, NH	1
13	Match .* greedily	.*Durham.*	1 Main St. Durham, NH	4
	Entire pattern matches!			

Code is **much** smarter, but results must follow this model

String Patterns

- What kinds of patterns might we want to define?
  - We surely want to specify how <u>characters</u> in the string match <u>characters</u> in the pattern
    - *exact character match*: match the letter “A”
    - *subset match*: match any letter, or match any digit: [a-z] or [0-9]
    - *“anything but”* match: any character that is <u>not</u> a space: [^☐]

☐ represents a space in the notes.

    - *wildcard match*: match any character: .
  - We might want to specify how many *replications* of a particular match are part of the pattern
    - 0 or more, 1 or more, 3 or more, between 2 and 5, etc.

Simple Examples

- . means any character
- * means 0 or more replications of preceding
- + means 1 or more replications of preceding

Pattern	**Meaning**	**Examples**
AAT	AAT	AAT
A.T	A then any 1 char then T	AAT, ATT, AGT, ACT, ...
AAT*	AA then 0 or more T's	AA, AAT, AATT, AATTT, ...
AAT+	AA then 1 or more T's	AAT, AATT, AATTT, ...
T+A*GG	1 or more T, then 0 or more A, then GG	TTAAAGG, TTTTTAGG, TTGG, TGG, ...
AA.+T	AA then 1 or more of anything then T	AAGT, AACGT, AAGAAT, AATT, AATTT, AATTTT, ...

These show **greedy** matching

Grouping

- (...) groups regular expression components
  - can specify a replication on groups

Pattern	Meaning	Examples
(AT)+	1 or more AT	AT, ATAT, ATATAT, ATATATAT, ...
TT(AT)*(CG)*	TT then 0 or more AT then 0 or more CG	TT, TTAT, TTCG, TTATCG, TTATATATCG, TTCGCGCG, ...
((AT)*(CG)*)+	1 or more of (AT)*(CG)*; i.e., 1 or more AT, CG pairs in any order	AT, CG, CGAT, ATCG, ATATATCG, CGATATCGAT, ...

Character Classes

Character class version is more efficient

- [...] defines a *character class*
  - a class represents <u>one</u> character that can be any class member
- \- defines a range of characters: a-z or 0-9

Pattern	Meaning	Examples
[ACTG]	A or C or T or G	A, C, T, G
(A|T|G|C)	A or C or T or G	A, C, T, G
[ACTG][ACTG]	(A or C or T or G) then (A or C or T or G)	AA, AC, AT, AG, CA, CC, CT, CG
[AC]*[GT]+	0 or more of (A or C) then 1 or more of (G or T)	ACAAGG, ACTT, AACGGG, AACG
[a-z]+	1 or more lower case letters	able, ace, abcdefghijklmnopqrstuvwxz
[a-zA-Z]+	1 or more letters	Able, aCe, abcdefghIJklmnopqrStuvWxz

Alternatives: |

- | (or) defines alternative patterns
  - the test string matches the (piece of) the pattern if one of the alternatives matches

<table>
<tr><th>Pattern</th><th>Meaning</th><th>Examples</th></tr>
<tr><td>(A|T|G|C)+</td><td>1 or more of A, T, C, G in any order</td><td>AT, CG, CGA, ATCC, CATA</td></tr>
<tr><td>(AT|CG)+</td><td>1 or more AT, CG pairs in any order</td><td>AT, CG, CGAT, ATCG, ATATATCG, CGATATCGAT</td></tr>
<tr><td>((AT)*(CG)*)+</td><td>1 or more of (AT)*(CG)*; i.e., 1 or more AT, CG pairs in any order</td><td>AT, CG, CGAT, ATCG, ATATATCG, CGATATCGAT</td></tr>
<tr><td>ATG(A|T|C|G)+(TAG|TGA)</td><td>start codon, then any dna, then a stop codon</td><td>ATGAATCGATCCGATGA</td></tr>
</table>

Character Classes with [^]

- [^ ...] defines a *character class* by what is <u>not</u> in it
  - all characters are in the class <u>except</u> the ones listed

□ symbol represents a space.

Pattern	Meaning	Examples
[^a-z]	All characters except lower case letters	A, C, T, G, %, 1, #, 7
[^ACTG][ACTG]	anything but A, C, T, and G, then (A or C or T or G)	aA, rG, 8T, tC, @A
[□]*[^□]+	0 or more blanks then 1 or more non-blanks	□Abc, Abc, □□□2dfbp/
□*[^□]+	Same as above	□Abc, Abc, □□□2dfbp/
[^a-z]+	strings with no l.c. letters	ABLE, A1, A+B=123, ^RT^, ...
[^0-9a-z]+	strings with no l.c. or digits	ABLE, A#, A+B=EE, ^RT^, ...

Note: ^ is not special except right after [

Metacharacters and Escapes

- . + * [and] have special meaning in regular expressions; they are called <u>metacharacters</u>
- Metacharacters are: . + * [ ] ( ) { } $ \ ? | - ^
- To use a metacharacter in a pattern, precede it with a \; \ is called the *escape* character

<table>
<tr><th>Pattern</th><th>Meaning</th><th>Examples</th></tr>
<tr><td>(□|\||*)+</td><td>string of □|, and *</td><td>| | * * * | | | | | | □ □ | | | | |</td></tr>
<tr><td>[□|*]+</td><td>string of □|, and *</td><td>| | * * * | | | | | | □ □ | | | | |</td></tr>
<tr><td>[ATCG\-]+</td><td>DNA sequence with missing nucleotides</td><td>ATC-GTACGGC--AATCG</td></tr>
</table>

Escape character use is context dependent.

The only “real” metacharacter for character groups is -.

Predefined Character Classes

- There are a bunch of predefined character classes represented by *char* where *char* is a single letter

Predefined class	**Equivalent**	**Meaning**
\d	[0-9]	A digit character
\D	[^0-9]	A non-digit character
\s	[□\t\n\r\f]	“white” space: blank, tab (\t), new line or line feed (\n), carriage return (\r), new page or form feed (\f)
\S	[^□\t\n\r\f]	A non-white space character
\w	[a-zA-Z_0-9]	A *word* character
\W	[^a-zA-Z_0-9]	A non-word character

Examples with Predefined Character Classes

Pattern	Meaning	Examples
[\w$]	word characters plus $; valid characters in Java id	a, b, c, z, 0, 1, $, _
[a-z_A-Z][\w$]*	valid Java identifier	Nim, _root, temp1, aPlayer
\s*\S+\s+\S+	optional white space (\s*) then a token (\S+) then required white space (\s+) and another token (\S+)	print□name ++□id%D
\s*\w+\s+\w+	same as above, except the tokens can only be composed of word characters	print□name add□name2

Boundary Markers

- Can match a boundary between 2 characters, rather than the character itself
  - ^ matches the start of the test string†
  - $ matches the end of the test string†
  - \b matches a word boundary

† Not really true in *multi-line* mode

Pattern	Meaning	Examples
^\s*\S+\s+\S+\s*$	optional space then token then space then token then space then end of input	print□name ++□id%D
^\s*\w+\s+\w+\s*$	same as above, except the tokens can only be composed of word characters	print□name add□name2

Quantifiers

- Replication specifications are called *quantifiers*
  - *Greedy* quantifiers keep trying to match until the match fails: return the match that finds the first occurrence of the pattern and that uses the <u>*maximum*</u> possible amount of the input
    - *   0 or more with a *greedy* match
    - \+   1 or more with a *greedy* match
    - ?   0 or 1 (an optional piece of the pattern)
    - {*n*}   exactly *n* replications
    - {*n*,*m*} at least *n*, no more than *m*
    - {,m}   no more than *m* (can be 0)
    - {n,}   at least n

Quantifier Examples

Pattern	Meaning	Examples
(AT){6,}	6 or more AT: an AT *microsatellite*	ATATATATATATATATAT
ATG[ATCG]{3}+(TAG|TGA)	start codon (ATG), then any number of codons (3 nucleotides), then a stop codon (TAG or TGA)	ATGAATCGATCCGACTGA
(\(\d\d\d\))?\d\d\d\-\d\d\d\d (\(\d{3}\))?\d{3}\-\d{4}	US phone number format with optional area code in ( )	862-3780, (603)862-1234

Two different regular expressions for the same logical pattern

Underlining used to show groups of 3 nucleotides

Reluctant Quantifiers

- *Reluctant* quantifiers stop trying to match as soon as the entire pattern is matched: try to match the first occurrence of the pattern with the <u>*minimum*</u> amount of input
  - *? 0 or more with a *reluctant* match
  - +? 1 or more with a *reluctant* match
  - ?? 0 or 1 (an optional piece of the pattern)
  - {*n*}? exactly *n* replications (same match as greedy)
  - {*n*,*m*}? at least *n*, no more than *m*
  - {,m}? no more than *m* (can be 0)
  - {n,}? at least n

Reluctant v. Greedy Quantifiers

Pattern	Input	Match	Notes
TA*	GGGTAAAAC	TAAAA	As many A as can
TA*?	GGGTAAAAC	T	0 A
(AT)+	GGATATATACTT	ATATAT	As many AT as can
(AT)+?	GGATATATACTT	AT	1 AT
(GC){2,5}	AGCTGCGCGCAAA	GCGCGCGC	4 GC
(GC){2,5}?	AGCTGCGCGCAAA	GCGC	2 GC
(GC){5}	AGCTGCGCGCGCAAA	GCGCGCGCAA	5 GC
(GC){5}?	AGCTGCGCGCGCAAA	GCGCGCGCAA	Exactly 5 is always 5
TATAC?	GGACATATACTT	TATAC	1 C
TATAC??	GGACATATACTT	TATA	0 C

Java Regular Expressions

- java.util.regex.Pattern - encapsulates the idea of a regular expression and provides some of the functionality associated with matching an input string to the regular expression.
- java.util.regex.Matcher - encapsulates the complete matching process and options

Pattern Class

- Create a *Pattern* object with the *static* method, *compile*:
  - Pattern myPat = Pattern.compile(regEx);
    where regEx can be a *String*, *StringBuffer*, or *CharBuffer*
- Apply the pattern to an input string as a *separator* for the remaining parts of the input string
  - String[] matches = myPat.split(inputString);
    - if pattern = “:” and
    - inputString = “able: baker: charlie: delta”
    - matches[]: [“able”, “baker”, “charlie”, “delta”]

Matcher

- The real power in Java regular expressions comes from the *Matcher* class.

Matcher created by pattern for a string

```
String dna = readNextSequence();
Pattern orf = Pattern.compile( "atg.{3}*(tag|tga" );

Matcher m = orf.matcher( dna );
while ( m.find() )
{
  System.out.println( m.group() );
}
```

First, search from start of input; if match, remember where ended; start next search from there.

The substring matching the pattern

Java Literals

- The Java compiler parses literal strings looking for its own “escape” specifications: \n, \t, \r etc.
    - It has very specific rules about what is allowed to follow \
    - This complicates specification of regular expressions as literal strings
        - Must “double-escape” some stuff.
        - Especially every use of \ has to be doubled to \\
        - So, “([ACTG]{2})\1{5,}” becomes “([ACTG]{2})\\1{5,}”
        - and “**`(.*),\s*(.*),`**” becomes “**`(.*),\\s*(.*),`**”

Groups

- A *group* is a portion of the pattern in parentheses ( )
- Each group has a unique index (n) based on *counting left parentheses from the left of the pattern starting at 1*
- As the *regex processor* is trying to match a string to the pattern, it saves the string matched so far by each *group*
- These are called *capturing* groups

Groups

```
String in = “Smith, John, 123 Thayer.”;
Pattern names = Pattern.compile( “(.*),\\s*(.*),” );
dMatcher m = names.matcher( in );
while ( m.find() )
{
   System.out.println( m.group() );
   System.out.println( m.group( 1 ) );
   System.out.println( m.group( 2 ) );
}
```

Entire match

group 1: Smith

group 2: John

System.out

Smith, John,
Smith
John

Pattern Match Substitution

- Can *replace* the matched region with something else
  - actually, copy preceding unmatched region to StringBuffer followed by replacement for matched region

```
# replace all occurrences of "hot" with "pot"
String in = "the hot shot had the first shot."
Pattern hot = Pattern.compile( "hot" );
StringBuffer out = new StringBuffer();
Matcher m = hot.matcher( in );
while ( m.find() )
{
  m.appendReplacement( out, "pot" );
}
m.appendTail( out );
```

Copies unmatched string from search start to start of match, then copies the replacement ("pot") for the match.

Copies what follows last match

Pattern Match Substitution Example

```
String in =
 "the hot shot had the first shot."
Pattern hot = Pattern.compile( "hot" );
StringBuffer out = new StringBuffer();
Matcher m = hot.matcher( in );
while ( m.find() )
{
  m.appendReplacement( out, "pot" );
}
m.appendTail( out );
```

no match	match	replace
the□	hot	pot
□s	hot	pot
□had the first s	hot	pot
.		

in:

out: the□ pot□spot□had□the□first□spot .

Groups with Replacement

```
String in = "Smith, John, 123 Thayer.";
Pattern names = Pattern.compile( "(.*),\\s*(.*)," );
StringBuffer out = new StringBuffer();
Matcher m = names.matcher( in );
while ( m.find() )
{
   System.out.println( m.group() );
   System.out.println( m.group( 1 ) );
   System.out.println( m.group( 2 ) );
   m.appendReplacement( out, "$2 $1," );
}
m.appendTail( out );
System.out.println( out )
```

Entire match

group 1: Smith

group 2: John

System.out

Smith, John,
Smith
John
John Smith, 123 Thayer.

Other Matcher Methods

- Some other useful Matcher methods:

int start(): index of start of match

int end(): index of <u>1 character after end of last match</u>

int start(int group): index of start of a group

int end(int group): index of <u>1 char after end of a group</u>

int groupCount(): number of groups matched

- There are more!

Match Flags

- Flags can control some match characteristics:
  - Pattern.DOTALL - normally . does not match line feeds
  - Pattern.MULTILINE - treat each line in input as separate input -- (^ matches start of line, $ matches end of line)
  - Pattern.CASE_INSENSITIVE - ignore case differences
  - Pattern.COMMENTS - white space is ignored in the pattern (all pattern white space has to be done with \s) and comments are allowed.

```
Pattern.compile( regex, Pattern.DOTALL | Pattern.MULTILINE  )
```

Comments

- Comments and spaces can be included in patterns
  - Must specify Pattern.COMMENTS "flag"
  - Comments start with "#" and end with "\n".
  - All white space and comments are deleted before compiling

```
String dna = readNextSequence();
Pattern orf = Pattern.compile(
          "ATG             # ATG is the start codon\n" +
          "( [ATGC]{3} )+  # then any number of codons\n" +
          "( TAG | TGA )   # ending with a stop codon ",
          Pattern.COMMENTS );

Matcher m = orf.matcher( dna );
while ( m.find() )
  System.out.println( m.group() );
```

```
Pattern.compile( "ATG([ATCG]{3})+(TAG|TGA)" );
```

Back Referencing

- A regular expression can include components that are *based on the input being processed*.
- This is done by *back referencing*
  - As the *regex processor* is trying to match a string to the pattern, it saves the string matched so far by each *group*
  - The pattern can include a reference to a previous group's matching string using the group's id, \1 or \2 or \3, etc.

Back Reference Example

- *Microsatellites* are regions in DNA that contain 6 or more exact copies of some sequence of nucleotides (A, C, T, G)
- a *dimer* microsatellite is 6 or more replications of 2 nucleotides
  - We could use a brute force r.e.: (AC|AT|AG|CT|CG|TG){6,}
    - this does not look for CA, TA, GA, TC, GC, or GT.
    - the approach would be impossible for longer base patterns.
  - Instead, we'll use back referencing

```
([ACTG]{2})\1{5,}
```

match \1 at least 5 more times, for a total dimer match >= 6

Group 1: matches any 2 nucleotides

\1 becomes whatever is matched by group 1

Dimer Matching Overview

Regex

([ACTG]{2})\1{5,}

Group 1

Input

AT**CACACACACACACA**AGCTAGC

- Regex processor key steps:
  - match group 1 to AT; this becomes \1
    \1 (AT) does **not** match CA, so fail

    ATCACACACACACACAAGCTAGC
  - match group 1 to TC, which becomes \1
    \1 (TC) does **not** match AC, fail

    A**TC**ACACACACACACAAGCTAGC
  - match group 1 to CA, which becomes \1
    \1 (CA) matches the next 6 pairs, which is 5 or more, so have

    AT**CA**CACACACACACAAGCTAGC

a complete match to the pattern

AT**CACACACACACACA**AGCTAGC

Dimer Matching Detail

Regex

([ACTG]{2})\1{5,}

Input

AT**CACACACACACACA**AGCT
AGCAGCAGCAAGCAACAACA
ACAACAACAGCACGTCGATG
AAGGAAGTCATAGCAGTTTC
AGCGACGCAGCGGCCCTTAA
TTTAGCAGAACGGTCGCCTC

Pattern Position	**Pattern Char**	**Input Position**	**Action**
([ACTG]{2})\1{5,}	[ACTG]	ATCACACA...	match A (1 of 2)
([ACTG]{2})\1{5,}	[ACTG]	*A*TCACACA...	match T(2 of 2); group 1 matches AT; \1 = AT
([ACTG]{2})\1{5,}	AT	*AT*CACACA...	C fails to match A, backup
([ACTG]{2})\1{5,}	[ACTG]	ATCACACA....	match T (1)
([ACTG]{2})\1{5,}	[ACTG]	A*T*CACACA...	match C(2); group 1 matches; \1=TC
([ACTG]{2})\1{5,}	TC	A*TC*ACACA...	A fails to match T, backup
([ACTG]{2})\1{5,}	[ACTG]	ATCACACA...	match C (1)
([ACTG]{2})\1{5,}	[ACTG]	AT*C*ACACA...	match A(2); group 1 matches; \1=AC

italic smaller font: characters matched so far

Dimer Matching (2)

Regex

([ACTG]{2})\1{5,}

Input

AT**CACACACACACACA**AGCT
AGCAGCAGCAAGCAACAACA
ACAACAACAGCACGTCGATG
AAGGAAGTCATAGCAGTTTC
AGCGACGCAGCGGCCCTTAA
TTTAGCAGAACGGTCGCCTC

Pattern Position	Pattern Char	Input Position	Action
([ACTG]{2})\1{5,}	[ACTG]	AT*C*ACACA...	match A(1); group 1 matches; \1=CA
([ACTG]{2})\1{5,}	CA	AT*CA*CACA...	match C
([ACTG]{2})\1{5,}	CA	AT*CAC*ACA...	match A, done with \1 match(#1)
([ACTG]{2})\1{5,}	CA	AT*CACA*CA...	match C
([ACTG]{2})\1{5,}	CA	AT*CACAC*A...	match A (\1 #2)
([ACTG]{2})\1{5,}	CA	AT*CACACA*CAC...	match C
([ACTG]{2})\1{5,}	CA	AT*CACACAC*AC...	match A (\1 #3)
([ACTG]{2})\1{5,}	CA	AT*CACACACA*C...	match C
([ACTG]{2})\1{5,}	CA	*CACACACAC*ACAC...	match A (\1 #4)
([ACTG]{2})\1{5,}	CA	*CACACACACA*CAC...	match C
([ACTG]{2})\1{5,}	CA	*CACACACACAC*AC...	match A (\1 #5)

Dimer Matching (3)

Regex

([ACTG]{2})\1{5,}

Pattern Position	Pattern Char	Input Position	Action
([ACTG]{2})\1{5,}	CA	*CACACACACAC*ACAA...	match A (\1 #5)
([ACTG]{2})\1{5,}	CA	*CACACACACACA*CAA...	match C
([ACTG]{2})\1{5,}	CA	*CACACACACACAC*AA...	match A (\1 #6)
([ACTG]{2})\1{5,}	CA	*CACACACACACACA*A...	C fails to match A but pattern is matched, no backup

Input

AT**CACACACACACACA**AGCT
AGCAGCAGCAAGCAACAACA
ACAACAACAGCACGTCGATG
AAGGAAGTCATAGCAGTTTC
AGCGACGCAGCGGCCCTTAA
TTTAGCAGAACGGTCGCCTC

Note that 7 copies of the dimer were matched, even though 6 was the minimum needed. This was a *greedy* match; it kept trying to match the pattern until it failed. You can specify a *reluctant quantifier* that quits matching as soon as pattern is satisfied.

Other Microsatellites

Regex for dimer microsatellite ([ACTG]{2})\1{5,}

- What regular expression would we use to find any *trimer* (3 nucleotide) microsatellite?

  ([ACTG]{3})\1{5,}

- What about a *tetramer* (4 nucleotides)?

  ([ACTG]{4})\1{5,}

- One regular expression to find <u>all</u> microsatellites:

  ([ACTG]{2,})\1{5,}

{2,}: all microsatellites of length greater than 2

Huntington's Disease

- Huntington's disease is a severe neurological disorder
  - symptom onset usually occurs between 30-50 years old
  - deterioration is progressive
  - although the cause is not fully understood, the condition can be identified by an excessive replication count of a *trimer* (3 nucleotides) microsatellite in a gene on chromosome 4
  - the "normal" gene has between 6 and 35 repeats of the trimer, CAG
  - Huntington's sufferers have 36 (or more) repeats
  - What is a regex to detect this?
    - (CAG){36,}

String vs. StringBuffer

- Pattern replacement methods use *StringBuffer*s, rather than *String*s because they are <u>much</u> more efficient
  - *String* objects are *immutable*; they cannot be changed

```
String results = "";
String[] parts;
parts = pattern.split( inputString );
for ( int i = 0; i < parts.size; i++ )
  results += parts[ i ] + "\n";
```

Java object creation is expensive!

Every execution of the loop body means the creation of a **new** *String* object whose value is the concatenation of 3 strings.

StringBuffer objects <u>can</u> be modified, so can do the same operation with **no** object creation, except for 1 *StringBuffer*.

StringBuffer

- *StringBuffer*s are dynamically sized
  - they have a current *length* and a current *capacity* where *length* <= *capacity*
- *StringBuffer* has many variations of the following mutation operations
  - append new characters to the end
  - insert new characters inside the existing string
  - delete characters from the string

StringBuffer append/insert

- The *append* and *insert* methods are overloaded so that any kind of primitive or *Object* can be converted to a string representation and added to the end of the current string; if necessary, the *capacity* is increased.

Add to the end of the current string.

```
append( String s );
append( float f );
append( double d );
append( char c );
append( char[] c );
append( byte b );
append( short s );
append( int i );
append( long l );
append( boolean b );
append( Object o );
// and a couple more
```

Insert converted object to start at position *p*

```
insert( int p, String s );
insert( int p, float f );
insert( int p, double d );
insert( int p, char c );
insert( int p, char[] c );
insert( int p, byte b );
insert( int p, short s );
insert( int p, int i );
insert( int p, long l );
insert( int p, boolean b );
insert( int p, Object o );
// and a couple more
```

Replacement and deletion

- There are several mechanisms for modifying or deleting portions of the current string:
  - deleteCharAt(int position);
  - delete(int start, int end);
  - setCharAt(int position, char newChar);
  - replace(int start, int end, String s);
    - the length of the s does **not** need to match the length of the string being replaced.

StringBuffer search

- *StringBuffer* has search methods:
  - int indexOf(String s)
  - int indexOf(String s, int fromIndex)
  - int lastIndexOf(String s)
  - int lastIndexOf(String s, int fromIndex)
- It does **not** have a *matches(String regex)* method

StringBuilder

- *StringBuffer* is a <u>thread-safe</u> class; you can use it in a multi-threaded environment
  - that makes it more robust, but slower
- If you are using a single-thread implementation, you can use *StringBuilder*; it has the same interface as *StringBuffer*, but is faster, but not thread-safe.

MacBookPro (2GB) performance:

14M appends, 7M deletions, and 7M *indexOf* with small strings.

	String	StringBuffer	StringBuilder
sec	5.25	3.23	2.06
[illegible]	[illegible]	[illegible]	[illegible]

Without *indexOf*:

	String	StringBuffer	StringBuilder
sec	4.74	2.37	1.16
[illegible]	[illegible]	[illegible]	[illegible]

18818 **cumulative** appends of 60 chars

	String	StringBuffer	StringBuilder
sec	168.43	0.24	0.24
ratio	**702**	1	1

Review

- Regular expressions are a powerful representation tool
- Java supports full regular expressions
  - character classes, both inclusion and exclusion
  - alternatives (|)
  - replications: *, +, { ... }, ?
  - greedy and non-greedy matching
  - back-referencing patterns
  - substitution

Next, in 416

- Sorting and searching algorithms