CS 251
Intermediate Programming
Regular Expressions

Brooke Chenoweth

University of New Mexico

Spring 2024

Regular Expressions

- A regular expression is a sequence of characters that forms a search pattern.
- Abbreviated *regex* or *regexp*
- Used to search, edit, manipulate text & data
- Search commands like `grep` use them
- Many programming languages provide regex support

Java Regular Expressions

- Supported by `java.util.regex`
- `Pattern` – compiled representation of regex
- `Matcher` – engine that interprets the pattern and performs the matches
- Can use these classes directly to manipulate text.
- Other classes use them "under the hood"
  - `String` has `matches` and `split` methods
  - `Scanner`'s delimiter is a `Pattern`

Simple examples

- `foo` – matches literal text `foo`
- [a-z]at – matches `bat`, `cat`, `hat`, `rat`, etc. A letter from `a` to `z` followed by `at`

String Literal

- Most basic pattern is a string literal.
- Match will find identical string.
- API also includes *metacharacters*, which have special meaning.
  - They are `<([{\^-=$!|]})?*+.>`
  - Force to be treated as ordinary character by preceding with a backslash.

Backslash mania

- Backslashes in regular expressions
- Backslashes within Java strings must be escaped.
- To match a literal backslash, you would use "\\\\"

Character Classes

- `[abc]` – a, b, or c (simple class)
- `[^abc]` – Any character except a, b, or c (negation)
- `[a-zA-Z]` – a through z, or A through Z, inclusive (range)
- `[a-d[m-p]]` – a through d, or m through p: `[a-dm-p]` (union)
- `[a-z&&[def]]` – d, e, or f (intersection)
- `[a-z&&[^bc]]` – a through z, except for b and c: `[ad-z]` (subtraction)
- `[a-z&&[^m-p]]` – a through z, and not m through p: [a-lq-z] (subtraction)

Predefined Character Classes

- `.` – Any character
- `\d` – A digit: `[0-9]`
- `\D` – A non-digit: `[^0-9]`
- `\s` – A whitespace character: `[ \t\n\x0B\f\r]`
- `\S` – A non-whitespace character: `[^\s]`
- `\w` – A word character: `[a-zA-Z_0-9]`
- `\W` – A non-word character: `[^\w]`

Quantifiers

Greedy	Reluctant	Possessive	Meaning
`X?`	`X??`	`X?+`	X, once or not at all
`X*`	`X*?`	`X*+`	X, zero or more times
`X+`	`X+?`	`X++`	X, one or more times
`X{n}`	`X{n}?`	`X{n}+`	X, exactly n times
`X{n,}`	`X{n,}?`	`X{n,}+`	X, at least n times
`X{n,m}`	`X{n,m}?`	`X{n,m}+`	X, at least n but not more than m times

Greedy, Relucant, and Possessive

- Greedy - attempt to match the entire string first, then match progressively shorter strings
- Reluctant - attempt to match as short a string as possible, then match progressively longer strings
- Possessive - attempt to match the entire string (once)

Boundary Markers

- `^` The beginning of a line
- `$` The end of a line
- `\b` A word boundary
- `\B` A non-word boundary
- `\A` The beginning of the input
- `\G` The end of the previous match
- `\Z` The end of the input but for the final terminator, if any
- `\z` The end of the input

Interactive Demo Program

```
public static void main(String[] args) {

  Scanner in = new Scanner(System.in);

  while(true) {
    System.out.print("Enter regex: ");
    String regexStr = in.nextLine();

    System.out.print("Enter test string: ");
    String testStr = in.nextLine();

    Pattern pattern = Pattern.compile(regexStr);
    Matcher matcher = pattern.matcher(testStr);

    boolean found = false;
    while (matcher.find()) {
      System.out.format("I found the text \"%s\" starting at " +
                        "index %d and ending at index %d.%n",
                        matcher.group(),
                        matcher.start(), matcher.end());
      found = true;
    }
    if(!found) { System.out.format("No match found.%n"); }
  }
}
```