

Regular Expressions

String Matching

- The problem of finding “a string that looks kind of like ...” is common.
 - e.g. finding useful delimiters in a file, checking for valid user input, filtering email, ...
- “Regular expressions” are a common tool for this.
 - Most languages support regular expressions.
 - e.g. in Java, they can be used to describe valid delimiters for `Scanner` (and other places).

Matching

- When you give a regular expression (or “regex”), you can check a string to see if it “matches” that pattern.
- e.g. suppose we have a regular expression to describe the “comma then maybe some whitespace” delimiters.
 - The string `","` would match that regex. So would `","` and `","\n"`.
 - But these wouldn't: `" , "` `" , , "` `"word"`

Note

- The “finite state machines” and “regular languages” from MACM 101 are closely related.
 - They describe the same sets of characters that can be matched with regular expressions.
 - (Regular expression implementations are sometimes extended to do more than the “regular language” definition.)

Basics

- When we specified a delimiter:

```
new Scanner(...).useDelimiter(",");
```

- ... the ", " is actually interpreted as a regular expression.
- Most characters in a regex are used to indicate “that character must be right here”.
 - e.g. the regex "abc" matches only one string: "abc"
 - Literal translation: “an ‘a’, followed by a ‘b’, followed by a ‘c’.”

Repetition


- You can specify “this character repeated some number of times” in a regular expression.
- e.g. match "wot" or "woot" or "wooot" or ...
- A * says “match zero or more of those.”
- A + says “match one or more of those.”
- e.g. the regex `wo+t` will match the strings above.
 - literal translation: “a ‘w’, followed by **one or more** ‘o’, followed by a ‘t’.”

Example

- Read a text file, using “comma and any number of spaces” as the delimiter.

```
Scanner filein = new Scanner(  
    new File("file.txt")  
).useDelimiter(", *");  
  
while ( filein.hasNext() ) {  
    System.out.printf("(%s)", filein.next() );  
}
```

a comma, followed by
zero or more spaces



Character Classes

- In our example, we need to be able to match “any one of the whitespace characters”.
- In a regular expression, several characters can be enclosed in [...].
 - That will match **any one** of the characters.
- e.g. The regex `a[123][45]` will match these:
 `"a14" "a15" "a24" "a25" "a34" "a35"`
 - “an ‘a’; followed by a 1, 2, or 3; followed by a 4 or 5”

Example

- Read values, separated by a comma, and one whitespace character:

```
Scanner filein = new Scanner(...)  
    .useDelimiter(", [ \n\r\t]");
```

- “Whitespace” technically includes some other characters, but these are the most common: space, newline, carriage return, tab.
 - `java.lang.Character` contains the “real” definition of whitespace.

Example

- We can combine this with repetition to get the “right” version.
 - a comma, followed by some (optional) whitespace

```
Scanner filein = new Scanner(...)  
    .useDelimiter(", [ \n\r\t]*");
```
- The regex matches “a comma followed by zero or more spaces, newlines, returns, or tabs.”
 - exactly what we were looking for.

More Character Classes

- A character range can be specified
 - e.g. `[0-9]` will match any digit.
- A character class can also be “negated”, to indicate “any character except”.
 - Done by inserting a `^` at the start.
 - e.g. `[^0-9]` will match anything except a digit.
 - e.g. `[^\n\r\t]` will match any non-whitespace.

Built-In Classes

- Several character classes are predefined, for common set of characters.
 - `.` (period): any character
 - `\d`: any digit
 - `\s`: any space
 - `\p{Lower}`: any lowercase character, `[a-z]`
- These often vary from language to language.
 - period is universal, `\s` is common; `\p{Lower}` is Java-specific (usually it's `[:lower:]`).

Examples

- `[A-Z] [a-z] *`
 - title-case words (“Title”, “I”; not “word” or “AB”).
- `\p { Upper } \p { Lower } *`
 - same as previous
- `[0-9] . *`
 - A digit, followed by anything (“5 q”, “234”, “2”).
- `gr[ea]y`
 - either “grey” or “gray”

Other Regex Tricks

- Grouping: parens can group chunks together.
 - e.g. `(ab)+` matches “ab”, “abab”, “ababab”, ...
 - e.g. `([abc] *)+` matches “a”, “a b c”, “abc ”, ...
- Optional parts: the question mark
 - e.g. `ab?c` matches only “abc” and “ac”
 - e.g. `a(bc+)?d` matches “ad”, “abcd”, “abccccccd”, but not “abd” or “acccccd”
- ... and many more options as well.

Other Uses

- Regular expressions can be used for much more than describing delimiters.
- The `Pattern` class (in `java.util.regex`) contains Java's regular expression implementation.
 - It contains static functions that let you do simple regular expression manipulation.
 - ... and you can create `Pattern` objects that do more.

In a Scanner...


- Besides separating tokens, a regex can be used to validate a token when it is read.
 - ... by using the `.next(regex)` method.
 - If the next token matches the regex, it is returned.
 - `InputMismatchException` is thrown if not.
- This allows you to quickly make sure the input is in the right form.
 - ... and ensures you don't continue with invalid (possibly dangerous) input.

Example

```
Scanner userin = new Scanner(System.in);  
String word;
```

```
System.out.print("Enter a word: ");  
try {  
    word = userin.next("[A-Za-z]+");  
    System.out.printf(  
        "That word has %d letters.\n",  
        word.length() );  
} catch (InputMismatchException e) {  
    System.out.println("That wasn't a word.");  
}
```

next token, but only if
it contains only letters.



Simple String Checking

- The `matches` function in `Pattern` takes a regex and a string to try to match.
 - returns a boolean: `true` if the string matches.
- e.g. previous example could be done without an exception:

```
word = userin.next();  
if( matches("[A-Za-z]+", word) ) { ... // a word  
} else { ... // give error message  
}
```

Compiling a Regex

- When you match against a regex, the pattern must first be analyzed.
 - The library does some processing to turn it into some more-efficient internal format.
 - It “compiles” the regular expression.
- It would be inefficient to do this many times with the same expression.

Compiling a Regex

- If a regex is going to be used many times, it can be compiled, creating a `Pattern` object.
 - It is only compiled when the object is created, but can be used to match many times.
- The function `Pattern.compile(regex)` returns a new `Pattern` object.

Example

```
Scanner userin = new Scanner(System.in);
Pattern isWord = Pattern.compile("[A-Za-z]+");
Matcher m;
String word;
System.out.println("Enter some words:");
do {
    word = userin.next();
    m = isWord.matcher(word);
    if ( m.matches() ) { ... // a word
    } else { ... // not a word
    }
} while( !word.equals("done") );
```

Matchers

- The `Matcher` object that is created by `patternObj.matcher(str)` can do a lot more than just match the whole string.
 - give the part of the string that actually matched the expression
 - find substrings that matched parts of the regex
 - replace all matches with a new string
- Very useful in programs that do heavy string manipulation.