CSC207 Week 9

Larry Zhang

Logistics

A2 Part 2 is out.

A1 Part 2 marks out on Peer Assessment, for remarking requests, contact Larry.

Today’s outline

- File I/O
- Regular Expressions

File I/O: read and write files

We use different kinds of **I/O Stream** classes to read and write files.

- **Byte Streams**: handle I/O of raw binary data.
- **Character Streams**: handle I/O of character data, automatically handling translation to and from the local character set.
- **Buffered Streams**: optimize input and output by reducing the number of calls to the native API.
- **Scanner**: allows a program to read and write formatted text.

Byte Stream

```
FileInputStream in = new FileInputStream("input.txt");
FileOutputStream out = new FileOutputStream("output.txt");

int c;
while ((c = in.read()) != -1) {
    out.write(c);
}
```

Reads and writes **one byte** at a time.

Character Stream

```
FileReader in = new FileReader("input.txt");
FileWriter out = new FileWriter("output.txt");

int c;

while ((c = in.read()) != -1) {
    out.write(c);
}
```

Reads and writes **one char (two bytes)** at a time.

Buffer Streams

The previous streams are **unbuffered streams**, i.e., each **read()** and **write()** triggers a disk access at lower level. This can be very **expensive (slow)**.

Buffer streams:

- for read: one disk access **reads a batch** of data from the disk to a memory area (call buffer), then Java read() gets data from the buffer.
- for write: Java write() writes to the buffer (memory); when the buffer is full, flush the buffer to disk (**batch write**).
- Much smaller number of disk access, much more efficient.

Buffered Streams

```
BufferedReader in = new BufferedReader(new FileReader("words.txt"));
```

Scanner

We've used scanner to handle user input from console (**System.in**), for files it's basically the same thing. In fact, the **System.in** and **System.out** are essentially files.

```
BufferedReader in = new BufferedReader(new FileReader("words.txt"));
Scanner s = new Scanner(in);
while (s.hasNextLine()) {
    String line = s.nextLine();
    if (line.startsWith("ab")) {
        System.out.println(line);
    }
}
s.close();
```

DEMO #1:

`FileIO.java`

More I/O streams

- Data streams: read/write, int, float, double, etc
  - https://docs.oracle.com/javase/tutorial/essential/io/datastreams.html
- Object streams: read/write (serialized) objects
  - https://docs.oracle.com/javase/tutorial/essential/io/objectstreams.html

Regular Expressions

Slides materials adopted from CSC207 Fall 2015 St George

What is a regular expression

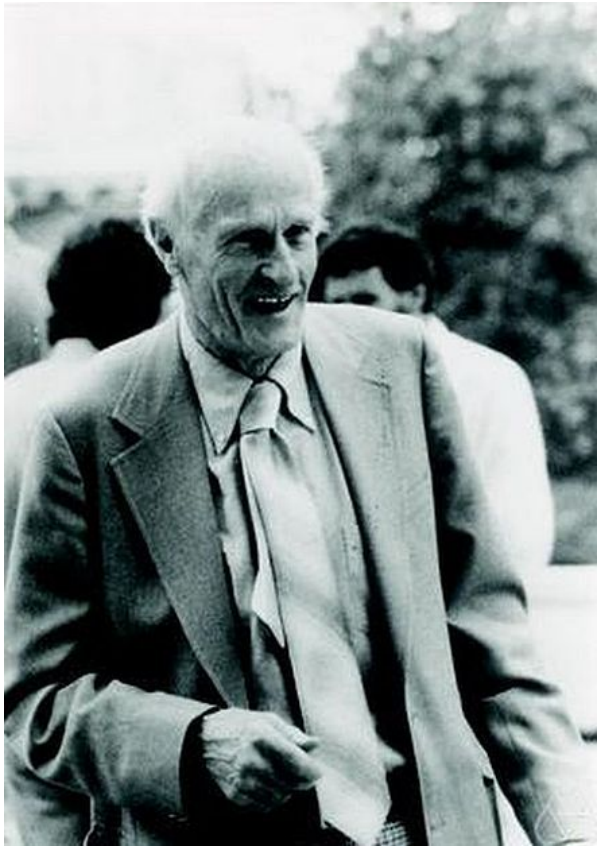
A **regular expression** is a pattern that a string may or may not match.

Example: `[0-9]+`

`[0-9]` means a character in that range
"+" means one or more of what came before

Strings that match: 9125 4
Strings that don't: abc empty string

Invented by
Stephen Cole Kleene



Example Uses

This regular expression `(\d{3}-)?\d{3}-\d{4}`

mathes phone numbers like “905-555-1234” or “555-1234”

This regular expression `[a-zA-Z]\d[a-zA-Z]\s\d[a-zA-Z]\d`

matches postal codes like L5L 1c6

Very compact and efficient way to do string matching, but can be evil unless you fully master how to use it properly.

A Regex for Validating RFC822 Email Addresses

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(
?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
\t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\0
31]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\
](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+
(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:
(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)
?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\
r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[
 \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)
?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t]
)*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
 \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*
)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)
*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+
|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r
\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:
\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t
]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031
]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](
?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?
:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?
:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?
:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|
\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>
@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"
(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?
:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[
\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-
\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(
?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;
:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([
^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\"
```

Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.

-- Jamie Zawinski

Simple Patterns

Pattern	Matches	Explanation
a*	” ‘a’ ‘aa’	zero or more
b+	‘b’ ‘bb’	one or more
ab?c	‘ac’ ‘abc’	zero or one
[abc]	‘a’ ‘b’ ‘c’	one from a set
[a-c]	‘a’ ‘b’ ‘c’	one from a range
[abc]*	” ‘acbccb’	combination

Anchoring

Lets you force the position of match

^ matches the beginning of the line

$ matches the end

Neither consumes any characters.

Pattern	Text	Result
b+	abbc	Matches
^b+	abbc	Fails (no b at start)
^a*$	aabaa	Fails (not all a's)

Escaping

Match actual

^ and $ and [etc.

using escape sequences

\^ and \$ and \[etc.

Remember, we also use escapes for other characters:

\t is a tab character

\n is a newline

Predefined Character Classes

Construct	Description
.	any character
\d	a digit [0-9]
\D	a non-digit [^0-9]
\s	a whitespace char [\t\n\x0B\f\r]
\S	a non-whitespace char [^\s]
\w	a word char [a-zA-Z_0-9]
\W	a non-word char [^\w]

Define Your Own Character Classes

Construct	Description
[abc]	a, b, or c (simple class)
[^abc]	any char 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 (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z except for b and c (subtraction)
[a-z&&[^m-p]]	a through z and not m through p (subtraction)

Quantifiers

Construct	Description
X?	0 or 1 times
X*	0 or more times
X+	1 or more times
X{n}	exactly n times
X{n,}	at least n times
X{n,m}	at least n but no more than m times

Capturing Groups

Capturing groups allow you to treat **multiple characters as a single unit**. Use parentheses to group.

For example, (BC)* means zero or more instances of BC, e.g., BC, BCBC, BCBCBC, etc.

Capturing Groups are Numbered

Capturing groups are numbered by counting their opening parentheses from left to right.

For example,
`((A)(B(C)))` has the following groups:

1. `((A)(B(C)))`
2. `(A)`
3. `(B(C))`
4. `(C)`

The numbers are useful because they can used as references to the groups.

Backreference

- The section of the input string matching the capturing group(s) is saved in memory for later recall via backreference.
- A backreference is specified in the regular expression as a backslash (\) followed by a digit indicating the number of the group to be recalled. For example,

Pattern	Example matching string
(\d\d)\1	1212
(\w*)\s\1	asdf asdf

Exercise

To match lines in a CSV file with format

```
Name,StudentNumber,mark
```

Examples:

```
sid smith,999912345,23

jane samantha doe,1000123456,100
```

Attempts

```
sid smith,999912345,23

jane samantha doe,1000123456,100
```

```
(.*),(.*),(.*)
(.*),(\d*),(.*)
(.*),(\d{9}|\d{10}),(.*)
(.*),(\d{9}\d?),(.*)
(.*),(\d{9}\d?),((100)|(\d\d))
(.*),(\d{9}\d?),((100)|(\d?\d))
(.*),(\d{9}\d?),((100)|([1-9]?\d))   // good
(.*),(\d{9,10}),((100)|([1-9]?\d))   // good
```

A given a set of strings that need to be matched,

a **proper regex** for this set

matches all strings in this set

and

does NOT match any string that is NOT in this set.

How to always come up with the proper regex?

Use Regex in Java

A little **DFA** is built here by the **compile** method.

```
Pattern p = Pattern.compile("CSC[0-9][0-9][0-9]H5(F|S)");
Matcher m = p.matcher("CSC207H5F");
System.out.println(m.matches());
```

```
System.out.println(Pattern.matches("a*b", "aaaaab"));
```

match() vs find()

- **match()** matches the **entire string** with the regex
- **find()** tries to find a **substring** that matches the regex

For example

```
Pattern p = Pattern.compile("\\d\\d\\d");
Matcher m = p.matcher("a123b");
System.out.println(m.find());    // true
System.out.println(m.matches()); // false
```

DEMO #2:

`RegexMatcher.java`

There are different “flavours” of regex

Different languages have different implementations of Regex which may behave differently.

http://www.greenend.org.uk/rjk/tech/regexp.html

Know your flavour before using it (read the manual).

References

Good tutorials:

- https://regexone.com/
- https://regex101.com/