Java II
Java Strings and Regular Expressions

Bernd Kiefer

Bernd.Kiefer@dfki.de

Deutsches Forschungszentrum für künstliche Intelligenz

String handling in Java

- The most basic implementation of strings: `char` arrays
  - \+ Can be modified
  - – No methods for manipulation, fixed length
- `String` class: a wrapper around `char` arrays

```
int length()
char charAt(int index)
int compareTo (String anotherString)
boolean contains(CharSequence s)
boolean equals(Object anObject)
String concat(String str)
boolean regionMatches(int toffset, String other,
                      int ooffset, int len)
String substring(intbeginIndex)
```

Modifying `String` *objects*

- `String` is the only class overriding operator `+`: contatenation

```
String firstName = "John", lastName = "Smith";
String fullName = firstName + lastName;
```

- `String` objects are *immutable*

```
String firstName = lastName = "Major";
firstName = "John";
```

- “modification” of a `String` always creates a new object!

Modifying `String` *objects II*

- Frequently modifying `String` objects may lead to inefficiency
- Run `concat.java` from the course home page as an example
- Alternative: `StringBuffer` class

- In General:

  Object creation and reclamation is costly: $\Longrightarrow$
  Avoid creating intermediate objects if not necessary!

Mutable `String`*:* `StringBuffer`

- `StringBuffer` objects are mutable (and thread-safe)
- Dynamic in size (in an efficient way)
- Modification and search methods
  - insert, append, delete, replace
  - indexOf, lastIndexOf
- More efficient single-thread version: `StringBuilder`, preferable in most cases

String Comparison

```
String s = "Ahab", t = "Ahab";
if (s == t)  System.out.println(s);
```

- Will s be printed?

String Comparison

```
String s = "Ahab", t = "Ahab";
if (s == t)  System.out.println(s);
```

- Will `s` be printed?
- No! operator == only tests for object identity!
- If comparison of strings is wanted, use `equals`

$$\texttt{x == y} \quad \Longrightarrow \quad \texttt{x.equals(y)}$$

$$\texttt{x.equals(y)} \quad \Longrightarrow \quad \texttt{x == y}$$

String Comparison

```
String s = "Ahab", t = "Ahab";
if (s == t)  System.out.println(s);
```

- Will `s` be printed?
- No! operator == only tests for object identity!
- If comparison of strings is wanted, use `equals`

$$\texttt{x == y} \quad \Longrightarrow \quad \texttt{x.equals(y)}$$

$$\texttt{x.equals(y)} \quad \not\Longrightarrow \quad \texttt{x == y}$$

Tokenization

- *Tokenization*: Split a string into several units (the *tokens*)
- Very frequently needed (where ?)
- class `java.util.StringTokenizer` provides this
  - give an input string and a set of delimiters
  - sequentially supplies tokens on request

```
StringTokenizer(String s, String del, boolean retDelims)
```

set of delimiters
default: " \t\n\r\f": (space, tab, newline, carriage-return, and the form-feed character)

Shall delimiters be tokens?

StringTokenizer *Example*

Split an input string using delimiters `,`, `(` and `)`, and let the delimiters themselves be tokens:

```
StringTokenizer st =
  new StringTokenizer("(1.0,2.0)", ",()", true);
while (st.hasMoreTokens())
  { System.out.println(st.nextToken()); }
```

Output:

```
(
1.0
,
2.0
)
```

`StreamTokenizer` *Class*

- Works on a `Reader` rather than a `String`
- More fine grained control over tokenization process:
  - Adaptable character set for words and white space
  - Optionally ignore C/C++ style comments
  - Are line breaks tokens or not
  - Optionally parses numbers
  - Support for quotation characters
- `Strings` can be processed by using `StringReader`
- `nextToken()` returns a token type rather than a string
- The string or number are in `sval` or `nval`, resp.
- Have a look at the API documentation!

StreamTokenizer *Example*

```
HashSet<String> wordlist = new HashSet<String>();
try {
  FileReader fr = new FileReader("InputFile.txt");
  BufferedReader br = new BufferedReader(fr);
  StreamTokenizer st = new StreamTokenizer(br);
  st.resetSyntax();
  st.wordChars('A', 'Z'); st.wordChars('a', 'z');
  while (st.nextToken() != StreamTokenizer.TT_EOF) {
    if (st.ttype == StreamTokenizer.TT_WORD)
      wordlist.add(st.sval);
  }
} catch (IOException ioex) { System.out.println(ioex); }
for (String s : wordlist) System.out.println(s);
```

This prints every word in `InputFile` exactly once.

Regular Expressions

- $\Sigma*$ is the set of all strings over alphabet Σ
- A regular expression r describes a set of strings $\mathcal{L} \in \Sigma*$
- $\mathcal{L}$ is called the language of r
- If a set $\mathcal{L}$ can be described by a regular expression, it is called a regular language
- An algorithm that checks if a string belongs to a regular language $\mathcal{L}$ is called recognizer or matcher
- Regular expressions are frequently used in string search and editing tasks (certainly in your favorite text editor, too)

Regular Expressions: Definition

- Regular expressions can be defined inductively:
  - Every element of Σ and ϵ (the empty string) is a regular expression
  - If α and β are regular expressions, so are
    - $(\alpha\beta)$ (concatenation)
    - $(\alpha \mid \beta)$ (alternative), and
    - $(\alpha*)$ (Kleene star: zero or more repetitions of α)
- Example: $((\mathrm{A} \mid (\mathrm{C} \mid (\mathrm{G} \mid \mathrm{T})))*)$ is the set of gene sequences of arbitrary length
- In real world systems: less brackets, lots of syntactic sugar like character classes or $+$ operator for *one* or more repetitions

Regular expressions in JAVA

`java.util.regex` package contains three relevant classes

- `Pattern` a compiled representation of a regular expression.
- `Matcher` the engine that interprets the pattern and performs match operations against an input string
- `PatternSyntaxException` an unchecked exception that indicates a syntax error in a regular expression pattern

Regular expressions: Code Example

```
String REGEX = "a(a|b)*b";
String INPUT = "aaabbb";
Pattern pattern;
Matcher matcher;
boolean found;
...
pattern = Pattern.compile(REGEX);
matcher = pattern.matcher(INPUT);
while(matcher.find()) {
  System.out.println("I found the text \"" +
  matcher.group() + "\" starting at " + matcher.start() +
  " and ending at " + matcher.end() + ".");
  found = true;
}
if(!found) { System.out.println("No match found."); }
```

Java Style of Regular expressions

- String literals: `REGEX = "john smith"`
- Metacharacters (characters with special meaning)
  - `.` (dot): matches any character
  - `^` and `$` match beginning/end of a string, respectively
  - Furthermore: `( ) [ ] { } \ | ? * +`
  - What if we need to match, e.g., `'['` literally?
    - precede metacharacter with backslash
    - everything enclosed in `\Q` up to `\E` is treated literally
    - Watch Out: a backslash in a Java `String` literal requires two: `"\\]"` `"\\Q[|]\\E"`

Character Classes & Quantifiers

- Character classes are abbreviations for sets of characters
- See `Pattern` API for specification of character classes

Character Classes & Quantifiers

- Character classes are abbreviations for sets of characters
- See `Pattern` API for specification of character classes
- metacharacters are different in character classes

Character Classes & Quantifiers

- Character classes are abbreviations for sets of characters
- See `Pattern` API for specification of character classes
- metacharacters are different in character classes
- Quantifiers:
  - x * : zero or more times x

Character Classes & Quantifiers

- Character classes are abbreviations for sets of characters
- See `Pattern` API for specification of character classes
- metacharacters are different in character classes
- Quantifiers:
  - x * : zero or more times x
  - x + : one or more times x

Character Classes & Quantifiers

- Character classes are abbreviations for sets of characters
- See `Pattern` API for specification of character classes
- metacharacters are different in character classes
- Quantifiers:
  - x * : zero or more times x
  - x + : one or more times x
  - x ? : zero or one time x

Character Classes & Quantifiers

- Character classes are abbreviations for sets of characters
- See `Pattern` API for specification of character classes
- metacharacters are different in character classes
- Quantifiers:
  - x * : zero or more times x
  - x + : one or more times x
  - x ? : zero or one time x
  - x{n,m} : n to m times x
  - x{n,} : at least n times x
  - x{,m} : at most m times x

Quantifier types

There are three sets of Quantifiers (? `*` `+` `{n,m}`)

- Greedy Quantifiers: tries to match as much as possible of the input string, reads the whole input prior to attempting the first match. Backs off one character, if the match fails and tries again.

Quantifier types

There are three sets of Quantifiers (? * + {n,m})

- Greedy Quantifiers: tries to match as much as possible of the input string, reads the whole input prior to attempting the first match. Backs off one character, if the match fails and tries again.
- Reluctant Quantifiers: starts at the beginning of the input string, then reluctantly eat one character at a time looking for a match.

Quantifier types

There are three sets of Quantifiers (? * + {n,m})

- Greedy Quantifiers: tries to match as much as possible of the input string, reads the whole input prior to attempting the first match. Backs off one character, if the match fails and tries again.
- Reluctant Quantifiers: starts at the beginning of the input string, then reluctantly eat one character at a time looking for a match.
- Possessive quantifiers always eat the entire input string, trying once (and only once) for a match. They never back off.

Quantifier Example

```
Current REGEX is: .*foo // greedy quantifier
Current INPUT is: xfooxxxxfoo
I found the text "xfooxxxxfoo" starting at 0 and ending at 11.

Current REGEX is: .*?foo // reluctant quantifier
Current INPUT is: xfooxxxxfoo
I found the text "xfoo" starting at 0 and ending at 4.
I found the text "xxxxfoo" starting at 4 and ending at 11.

Current REGEX is: .*+foo // possessive quantifier
Current INPUT is: xfooxxxxfoo
No match found.
```

Grouping

- Expressions can be grouped using parentheses:
  `((ab)*(b+(c)))`

Grouping

- Expressions can be grouped using parentheses: `((ab)*(b+(c)))`
- Such groups are by default *capturing*, i.e., the material in the group is saved in memory for later use

Grouping

- Expressions can be grouped using parentheses: `((ab)*(b+(c)))`
- Such groups are by default *capturing*, i.e., the material in the group is saved in memory for later use
- Capturing groups are numbered by counting opening parentheses from left to right:
  1. `((ab)*(b+(c)))`
  2. `(ab)`
  3. `(b+(c))`
  4. `(c)`

Using Capturing Groups

- Groups can be used with *back reference*: `"((ab)*)\\1"`
- Specify with a backslash (\) followed by a number
- Remember: you have to use two backslashes in a Java `String` literal to get one in the string
- The reference with number n has to match exactly the same string as was matched by group n
- the `group(int which)` method of class `Matcher` can be also be used to retrieve the matched groups.
- This allows to get many matches with a single `matches` call: a very specialized split

Splitting using RegExps

- The `split` method can be used to split a string with delimiters specified as regular expressions.

```
REGEX = ":++";
INPUT = "one::::two::three:four:five";
Pattern p = Pattern.compile(REGEX);
String[] items = p.split(INPUT);
for(int i = 0; i < items.length ; i++)
  { System.out.println(items[i]); }
```

- There are also `split` and `matches` methods in the `String` class for one-shot application of regular expressions
- Think about the creation of intermediate objects when using them

Replacement methods

- There are several methods to replace matched string portions by new material
- Again, there are convenience methods in `String`, too
- Some examples:

```
String replaceAll(String replacement)
String replaceFirst(String replacement)
Matcher appendReplacement(StringBuffer sb, String repl)
StringBuffer appendTail(StringBuffer sb)
```

appendReplacement *Example*

```
Pattern p = Pattern.compile("cat");
Matcher m = p.matcher("one cat two cats in the yard");
StringBuffer sb = new StringBuffer();
int i = 1;
while (m.find())
  { m.appendReplacement(sb, "dog" + i++); }
m.appendTail(sb);
System.out.println(sb.toString());
```

returns: `"one dog1 two dog2s in the yard"`

Further Reading to Java regular expressions:

Mastering Regular Expressions, 2nd Edition,
Jeffrey E.F. Friedl, O'Reillly, 2002