Java By Abstraction: Chapter 6

Strings

Some examples and/or figures were borrowed (with permission)
from slides prepared by Prof. H. Roumani

What are Strings?

- Sequence of characters
- Non-primitive (i.e., object) data type
- Read-only objects (recreated but not modified)
  - Any “changes” are actually new objects initialized with the new value

The Masquerade

- Remember, Strings are objects
- Strings can be initialized like objects:

  String name = new String("My name is Steven");
- But Strings can also be initialized like primitives:

  String name = "My name is Steven";
- The compiler replace the "short form" with the proper (i.e., object) initialization statement

Concatenation

- Strings can be joined using “+” operator

  String s = “EECS” + “1020”;

- Again, this is just a short form

- Compiler replaces with proper form

  String s = new String(“EECS1020”);

Character Indexing

- Indicate position within a String
- Numbered from 0 to length-1

```
String: EECS1020
        --------
Index:  01234567
```

Accessors

- Section 6.2.2
- Noteworthy methods:
  - length(): returns the number of characters in String
  - charAt(*index*): returns the char at the passed index
  - substring(*start, end*): returns a new String containing only the characters at the index from *start* (inclusive) to *end* (exclusive)

Transformers

- Section 6.2.3
- Noteworthy methods:
  - trim(): returns a new String with the same characters, but without leading and trailing whitespace

```
String text = "      extra space      ";
output.print(text.trim()); // outputs
  "extra space"
```

Comparators

- Section 6.2.4
- Noteworthy methods:
  - equals(*otherString*): returns true iff the two Strings are identical (see also equalsIgnoreCase(*otherString*))
  - indexOf(*otherString*): returns the index of the first occurrence of *otherString* in the String object; returns -1 if not found
  - compareTo(*otherString*): (see next slide)

s1.compareTo(s2) (in general)

- Assume *s1* and *s2* are both in lowercase (or both uppercase)
- Assume lexicographic (i.e., dictionary) ordering

- If *s1* and *s2* are identical, return value == 0
- If *s1* comes before *s2*, return value < 0
- If *s1* comes after *s2*, return value > 0

s1.compareTo(s2) (more specifically)

- Case 1: *s1* and *s2* are identical
  - Return: 0
- Case 2: one String starts with the other (e.g., *s1* = "Planet", *s2* = "Pl")
  - Return: *s1*.length() – *s2*.length()
- Case 3: there is a miss-match between *s1* and *s2* at some index, *k* (e.g., *s1* = "Planet", *s2* = "Pluto")
  - Return: *s1*.charAt(*k*) – *s2*.charAt(*k*) // subtract Unicode values

Strings ↔ Numbers

- Numbers → Strings:
  - “” + *number*
- Strings → Numbers:
  - “Wrapper” classes contain methods for handling primitive types (e.g., Integer, Double)
  - int num = Integer.parseInt(“514”);
  - double num = Double.parseDouble(“3.141592”);

Application: Character Frequency

- How many times does a character appear in a String?
  - Use charAt() method to access characters
  - Use a for loop to iterate over the string length
  - Increment a count if the character is found

Exercise: SentenceCounter

- Task:
  - Output the number of tokens that end with "."
- Code:
  - (Presented in lecture)

Application: Fixed-Size Codes

- Lookup value in one String, replace with value in a second String at same index
  - Use parallel strings for lookup
    - `0  1  2  3  4  5  6`
    - `Sun Mon Tue Wed Thu Fri Sat`
  - Use indexOf() method to find index of value in “top” String
  - Use substring() method to retrieve value from “bottom” String

Exercise: DigitSpeller

- Task:
  - Output numbers as words
  - E.g., “123” returns “onetwothree”
- Code:
  - (Presented in lecture)

Regular Expressions

CHARACTER SPECIFICATIONS	
`[a-m]`	*Range. A characters between a and m, inclusive*
`[a-m[A-M]]`	*Union. a through m or A through M*
`[abc]`	*Set. The character a, b, or c*
`[^abc]`	*Negation. Any character except a, b, or c*
`[a-m&&[^ck]]`	*Intersection. a though m but neither c nor k*
PREDEFINED SPECIFICATIONS	
`.`	*Any character*
`\d`	*A digit, [0-9]*
`\s`	*A whitespace character, [\t\n\x0B\f\r]*
`\w`	*A word character, [a-zA-Z_0-9]*
`\p{Punct}`	*A punctuation,* `[!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~]`
QUANTIFIERS	
`x?`	*x, once or not at all*
`x*`	*x, zero or more times*
`x+`	*x, one or more times*
`x{n,m}`	*x, at least n but no more than m times*

Exercise: PostalCodeChecker

- Task:
  - Output whether or not a String is a valid postal code
  - E.g., "M3J 1P3" returns `true`
- Code:
  - (Presented in lecture)

StringBuffer and StringBuilder

- Strings cannot be modified (no mutator methods)
- Repeatedly creating new Strings is inefficient
- StringBuffer allows char sequence modification
- StringBuffer mutator methods:
  - append: adds parameter to the end of the sequence
  - insert: adds parameter to this sequence at specified index; existing characters are shifted to the right
  - delete: removes characters between two indexes; existing characters are shifted to the left

StringBuilder

- Mutable like StringBuffer
- Newer than StringBuffer
- Slightly faster than StringBuffer because it is not “synchronized”
  - Example shown in lecture