CSE 1710

Lecture 17

Text, Strings (II)

Goals/To do:

Given a string and a character, derive the **frequency of the character within the string**

Given a string, a target character and a replacement character, **implement character substitution.**

Given a numeric value in string format, **parse into numeric type**

Goals/To understand:

- difference between `char`, `String`, and `StringBuffer`
- The non-primitive String masquerades as a primitive type
- Pattern-matching abstractions (regular expressions)
- The difference between raw and formatted text; how to separate content from presentation

Recap: Strings are “objects with benefits”

- Creating strings is **not different** from creating any other object
  - A String object, like any other object, has a state
  - the state of a string object: the sequence of characters that is encapsulated
- However, string objects have some bonus features
  - **they can *masquerade* as primitive value**
  - **they are efficient (but in exchange they are *immutable*)**
- masquerade aspect #1
  - string objects can be specified using literal-like syntax
    - `String s = “hello”;` *(** creation of new objects only conditionally)*
    - `System.out.println(“hello world”);`
- masquerade aspect #2
  - string objects can participate in expressions just like primitive-value operands
    - `“hello” + 89`

How to get String object from anything

- any object has `toString()` method
  - this also includes `String` objects, in which case `toString()` is redundant
- do primitive values have a toString() method?
  - no
  - so how do we transform?
    - concatenate primitive value to the empty string
    - `String str1 = “” + 9;`
    - `String str2 = “” + ‘x’;`

How to get primitive values from String objects

- suppose we have a sequence of characters
- suppose that sequences happens to be the same as a literal value from a primitive type
  - e.g., `“897”` `“875l”` `“false”` `“C”`
- Use any of these static methods
  - `Integer.parseInt(str)`
  - `Short.parseShort(str)`
  - `Byte.parseByte(str)`
  - `Long.parseLong(str)`
  - `Double.parseDouble(str)`
  - `Float.parseFloat(str)`
  - `Boolean.parseBoolean(str)`
- look at API, note the contract re: parameter
  - `java.lang.NumberFormatException: Value out of range.`

L17App1b

L17App1c

How to get primitive values from String objects

- suppose we have a one-character String and we want the corresponding char
  - e.g., `“C”` `“d”` `“9”`
- there is a wrapper class `Character`(just like the others)
- unfo, there is no `Character.parseCharacter(str)` or other such static method
- instead:
  `char c = “C”.charAt(0)`

String methods, recap

assume `str1`, `str2` are strings; `idx1`, `idx2` are integers

- `str1.length()` returns an `int`
  - tells us the number of characters in the object's character sequence
- `str1.charAt(idx1)` returns a `char`
  - gives us the character at the specified index
  - remember the first character of a string that is n characters long is at index 0 and the last character is at index n-1
- `str1.equals(str2)` returns a `boolean`
  - tells us whether str2 has the same state as str1
  - not whether str2 is the same object as str1
- `substring(idx1,idx2)` returns a `String`
  - gives a subset of the character sequence from the start index inclusive to the end index exclusive

String methods, recap

- `str1.compareTo(str2)` returns an `int`
  - gives us an `int` that is a coded message
    - 0 if if `str1` and `str2` are equal
    - polarity (the sign, +ve or –ve) tells us whether `str2` comes before `str1` in the dictionary.
    - dictionary uses lexicographic ordering
  - if `str1` and `str2` are not equal, then the value is Unicode difference of the first differing character
  - if there is no index position at which they differ, then the value is the length difference

String methods, some new ones

assume `str1`, `str2` are strings; `idx1`, `idx2` are integers

- `str1.toUpperCase()` returns a `String`
- `str2.toLowerCase()` returns a `String`
  - these are **NOT** mutators!!!
  - each returns a `String` obj, which is an entirely new object that is modified version of `str1`
  - `str1` is not changed at all (in fact, it **cannot** be changed, since it is immutable)
- `str1.substring(idx1)` returns a `String`
  - just like `str1.substring(idx1, idx2)`, with the assumption that `idx2` is the length of `str1`
  - anything you do using `str1.substring(idx1)`, you could also do with `str1.substring(idx1, idx2)`
  - CONVINCE YOURSELVES OF THIS

String methods

- `str1.indexOf(str2)` returns an `int`
  - if `str2` **does not** occur within `str1`, the method gives us the value `-1`
  - if `str2` **does** occur within `str1`, the method gives us a value which is the index at which `str2` occurs in `str1`'s character sequence
    - if `str2` occurs more than once within `str1`, the method gives us a value which is the index at which `str2` **first** occurs in `str1`'s character sequence
- `str1.indexOf(str2, idx1)` returns an `int`
  - just like `str1.indexOf(str2)`, but the methods looks at `str1`'s character sequence only starting at index position `idx1` onwards

Comparing strings: `equals` vs `matches`

suppose we have two strings, `str1` and `str2`

- `str1.equals(str2)` returns true iff
  - `str1` has the ***same state*** as `str2`
- `str1.matches(str2)` returns true iff
  - `str2` ***matches the pattern*** as stipulated by `str2`
  - in this context (i.e., being a parameter to `matches`)
    - str2 is interpreted as a **regular expression**

`“hello”.matches(“hello”)`	
REGEX criteria	`“hello”` **satisfies?**
the character **`h`** is in index position 0	yes
the character **`e`** is in index position 1	yes
the character **`l`** is in index position 2	yes
the character **`l`** is in index position 3	yes
the character **`o`** is in index position 4	yes
no further characters in the sequence	yes

Regular expressions: Simple classes

- a regular expression can also use ***special characters and syntax*** to specify more patterns more generally
- `[abc]` defines a simple class of characters

L17App2

`“hello”.matches(“[Hh]ello”)`	
REGEX criteria	**str1 satisfies?**
the character **`H`** or **`h`** is in index position 0	yes
the character **`e`** is in index position 1	yes
the character **`l`** is in index position 2	yes
the character **`l`** is in index position 3	yes
the character **`o`** is in index position 4	yes
no further characters in the sequence	yes

Regular expressions: Simple classes using a range

– [a-d] defines a simple class using a range

L17App3

“hello”.matches(“[a-d]ello”)

REGEX criteria	str1 satisfies?
the character **a** or **b** or **c** or **d** is in index position 0	yes
the character **e** is in index position 1	yes
the character **l** is in index position 2	yes
the character **l** is in index position 3	yes
the character **o** is in index position 4	yes
no further characters in the sequence	yes

Regular Expressions

– `[a-d[f-h]]` matches
  - any of `a`,`b`,`c`,`d`,`f`,`g`,`h`
  - the union of `a-d` and `f-h`

L17App4

– `[^a-d]` matches
  - any character that is NOT `a`, `b`, `c`, `d`,

L17App5

– `\d` matches any digit
  - same as: `[0-9]`

– `\s` matches any whitespace character:
  - same as: `[ \t\n\x0B\f\r]`
  - *vertical tab* is `\xOB`, aka `\u000B`

L17App6

– `\w` matches any word character:
  - same as: `[a-zA-Z_0-9]`

L17App7

Regular Expressions

- `a*` matches
  - zero or more `a`’s
- `a+` matches
  - 1 or more `a`’s
- `a?` matches
  - 0 or 1 `a`’s
- `a{n,m}` matches
  - at least n a’s but not more than m `a`’s

Regular Expressions

suppose we prompt the user for a time, with the instructions that the time must be one of 3, 6, or 9 am or pm

- acceptable: 9 am, 3 pm
- not acceptable: 10 am, 3 um, 9am, 9:00 am

– construct a regex to match this

- `“[369] [ap]m”`

suppose we want to allow the space to be optional

- acceptable: 9am, 12 am, 12pm
- not acceptable: 10am, 9:00am

– construct a regex to match this

- `“[369] ?[ap]m”` or `“[369][ ]?[ap]m”`