CSE 1710

Lecture 14, 15
String Handling

Strings

We have covered three chunks of material:

Week 1:

- "String Literals" pp. 22-23; Fig 1.12; PT 1.8

Week 6:

- "The String Class" Section 6.1.1, pp. 219-220
- "The Masquerade and the + Operator" Section 6.1.2, pp. 221-224

Today

- "The String Class" Section 6.2 pp. 219-220

String Methods

- length()
- charAt(int)
- substring(int,int) (int)
- indexOf(String) (String,int)
- toString() and equals()
- compareTo()
- toUpperCase() and toLowerCase()

REMEMBER!

– Any string is represented by **an object**

– A variable of type `String` is used to store **the address** of the object.

– The `String` object has a **state**
  - the **state** of an object is defined as **the value of all its attributes**
  - the **only attribute** of a `String` object is the attribute that represents the sequence of characters
  - the state of a `String` object basically boils down to **what is its sequence of characters?**

REMEMBER!

– If the state of a `String` object is such that its **sequence has no characters at all,** how do we understand this?
  - this is the *empty string*
  - the string has length **zero**
  - ***THIS IS NOT A NULL STRING***

  - **What is this "null string"?**
    - technically speaking, "null string" is not really a correctly-formed term, there is no such thing
    - HOWEVER, it is often used to mean a **string reference** that is set to `null`.
    - This means that a String reference has been declared, but that there is NO `String` object.

Can we modify the state of a `String` object?

– NO

– Once a string object is created, it cannot be changed.
  - This is called *immutability*
  - Strings are *immutable*

– This is an unusual property – MOST other objects are mutable

But what if we need to modify the state of a `String` object?

Instead of modifying the sequence, we just create new strings that are modified verisons of the originals.

– It is fast and easy, thanks to the + operator

– Given this, is it correct to say that String has mutators?
  - not technically; they are actually *generators of new modified objects*

`int : length()` method

- `str1.length()` returns an `int`
  - tells us the number of characters in the object's character sequence

`char : charAt(int)` method

- remember – the indexing of the character positions starts at 0!

- `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

`String : substring(int, int)` method
`String : substring(int)` method

what do each of these methods do?

these methods must return a brand new string

- `substring(idx1,idx2)` returns a `String`
  - gives a subset of the character sequence from the start index **inclusive** to the end index **exclusive**

Can you live w/o substring(int) given the overloaded (int,int)?

`int : indexOf(char)` method
`int : indexOf(char, int)` method

what do each of these methods do?

- `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

- `str1.substring(idx1)` [REVISITED]
  - just like `str1.substring(idx1, idx2)`, with the assumption that `idx2` is the length of `str1`

`int : indexOf(char)` method
`int : indexOf(char, int)` method

How would use use indexOf to detect <u>all</u> occurrences of a substring?

- `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 : toString()` method
`boolean : equals(String)` method

Do not underestimate what equals does

- `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

String matching/comparison (basic)

Suppose `c1, c2` are chars

Suppose `s1, s2` are Strings

- what does the equality boolean operator == tell us?
  - `boolean isMatch = c1==c2;`
  - `boolean isMatch = s1==s2;`
- what does `.equals(String)` tell us?
  - `boolean isMatch = s1.equals(s2);`
- what does `.compareTo(String)` tell us?
  - `int differingIndexPos = s1.compareTo(s2);`

`int : compareTo(String)` method

Elaboration of "`compareTo(String)`"

(sort of) *"tell me whether the passed string comes before this string in the dictionary"*

"aardvark".compareTo("anvil")

- *anvil* does not come before *aardvark* in the dictionary, so the result is no (negative value)

"anvil".compareTo("aardvark")

- *aardvark* **does** come before *anvil* in the dictionary, so the result is yes (positive value)

(better) *"tell me whether the passed string comes before this string in the dictionary and, for the first character that is the determining factor, what is the distance"*

- the second character is the determining factor ('a' vs 'n', there is a distance of 13 between them)

— `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 : toUpperCase()` method
`String : toLowerCase()` method

these methods must return a brand new string

— `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)

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`

- FOR NOW, WE WILL DO **DEAD SIMPLE** PATTERNS

`"hello".matches("hello")`

- in the context of being a parameter to `matches`, `str2` is interpreted as a **regular expression (aka REGEX)**
- the REGEX specifies 5 criteria:

`"hello".matches("hello")`	
REGEX criteria	**Criterion satisfied?**
that the character **`h`** is in index position 0	yes
that the character **`e`** is in index position 1	yes
that the character **`l`** is in index position 2	yes
that the character **`l`** is in index position 3	yes
that the character **`o`** is in index position 4	yes
(implied) 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:

L17App6

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

L17App7

  - same as: `[a-zA-Z_0-9]`

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"`

Numeric Strings – Using the Wrapper Classes

```
String s = "1020";

int n1 = Integer.parseInt(s);

long n2 = Long.parseLong(s);

double n2 = Double.parseDouble(s);

float n3 = Float.parseFloat(s);
```

number to string conversions? best handled using the + operator

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)`

L17App1b

L17App1c

– look at API, note the contract re: parameter

- `java.lang.NumberFormatException: Value out of range.`

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)`