Computer Science 252
Problem Solving with Java
The College of Saint Rose
Spring 2015

Topic Notes: Strings

This semester we've spent most of our time on applications that are graphical in nature:

- Manipulating graphical objects
- Implementing animation

Many practical programs, however, are not graphical in nature. Many are textual, and require some of the mechanisms for text processing. Much of what is in these notes is review of things you have seen earlier this semester or in previous coursework. Please read through these notes and don't hesitate to ask questions if anything is unclear.

Java's `String` class

Where have we seen examples of the Java `String` class?

- We can define variables and parameters of type `String`. You can assign values to `String` names, and you can write methods that return `String`s as results. Recall from the "drag students" example that we declared an array of `String`s representing file names of your images.
- We can specify `String` constants in programs by placing double quotes around a sequence of characters:

```
msg.setText("Got me!");
```

- We can concatenate (glue together) `String`s; or we can glue together a `String` and a numeric value to make a new `String`:

```
mouseCount.setText("You clicked " + count + " times.");
```

- We can determine if two `String`s look the same by using the `equals` method. We saw this, for example, when looking up which entry is currently selected in a `JComboBox`.

  It is important to remember to use `equals` to check for equality, rather than == or you will not get your expected result. == checks if we are looking at the exact same `String` (in the computer's memory) rather than two `String` objects which happen to have the same contents.

  Recall that we saw this same idea when comparing `Location` objects.

Uniqueness of Java's `String`s

Most Java types we've seen fall neatly into one of two categories:

- *Object types* (*e.g.*, `FilledRect`, `Text`, `Scanner`)
  - Described by a class definition
  - Created using constructor and `new` command
  - Manipulated through invocation of methods
  - Can be placed into an `ArrayList`
- *Non-object (*i.e.*, base, or primitive) types* (*e.g.*, `int`, `boolean`, `double`)
  - Members of each of these types are described by constants such as 3, true, 49.7
  - Manipulated through operator symbols like +, &&, *
  - Can be placed in arrays, but must be "wrapped" in "capital letter" classes to be placed in an `ArrayList`

Java considers `String` to be an object type. But `String`s share features with both object and non-object types.

- Can be described by constants (`"hello"`), like base types
- Can be manipulated using an operator symbol (+), like base types
- Can be created using a constructor and `new` command like objects
- Can be manipulated by invoking methods, like objects. In fact, there are over 50 methods in the `String` class!

Some useful `String` methods

- `public int length()`
  returns the length of a string.

```
String courseName = "Data Structures";
int nameLength = courseName.length();
```

`nameLength` gets the value 15.

- `public int indexOf(String s)`
  returns the starting index of the first occurrence of string `s`; returns `-1` if `s` was not found.

  ```
  courseName.indexOf("S")
  ```

  returns the value 5. Note that indexing starts at 0, as it does with arrays.

  Of course, we can also say

  ```
  String searchString = "S";
  ```

  and then

  ```
  courseName.indexOf(searchString)
  ```

  Another option is:

  ```
  public int indexOf(String s, int startIndex)
  ```

  which begins its search at the specified index.

  What are the results of each of the following?

  ```
  courseName.indexOf("Struct", 2)
  courseName.indexOf("Struct", 8)
  courseName.indexOf("t", 2)
  courseName.indexOf("t", 6)
  courseName.indexOf("t", 7)
  ```

  Answers: 5, -1, 2, 6, 10.

  Here is a method that searches for the number of occurrences of a `String`, `word`, in another `String`, `text`.

  ```
  public int wordCount(String text, String word) {
     int count = 0;
     int pos = text.indexOf(word,0);
     while (pos >= 0) {
        count++;
        pos = text.indexOf(word,pos+word.length());
     }
     return count;
  }
  ```

  What does the above method return for these call?

```
wordCount("yabbadabbadoo","abba");
wordCount("scoobydoobydoo","oo");
```

How about this one? [Side note: I bet everyone knows where the strings above come from but what about this next one?]

```
wordCount("bubbabobobbrain","bob");
```

Is that what you want? How would you modify the method to include the overlapped "bob"s above?

- Converting strings

  Note that `"s"` and `"S"` are different strings.

```
public String toLowerCase()
public String toUpperCase()
```

  A case-insensitive word counter/finder differs from the above code only by the added two lines in the following.

```
private int substringCounter( String text, String word)
{
      int count = 0;
      int pos;

      text = text.toLowerCase();
      word = word.toLowerCase();

      pos = text.indexOf(word,0);

      while ( pos >= 0 )
      {
          count++;
          pos = text.indexOf(word,pos+word.length());

      }
      return count;
}
```

  Note that the assignment statements to text and word are required. `String` methods do not manipulate the given `String`. They make a brand new one. We say that Java `String`s are *immutable*.

- "Cutting and pasting" strings

- To paste together two strings, use "+"
- To cut (i.e., to extract a substring of a string) use:

```
public String substring(int startIndex, int indexBeyond)
```

For example:

```
String countText = "3 Balls, 2 Strikes, 2 Outs";
String strikesOnly;
strikesOnly = countText.substring(9,18);
```

Note that if we omit the second parameter, the substring returned is the one starting at `startIndex` and will include the rest of the string.

Continuing the example above:

```
String outsOnly;
outsOnly = countText.substring(20);
```

Link finder:

We want to write a program to find and extract all of the links in an HTML file. To do this, we need to know how a link is defined in an HTML file:

```
<a href="the URL">link </a>
```

So we need to find the tags "<a>" and "" that surround the URL.

Convert the string that is the HTML file to lowercase.

```
String links = "";
```

Find the first position of "<a"

While there is a link remaining (i.e., `tagPos` is not 1)

- Find the other end of it - i.e., ">"
- Get the substring between those (that's the URL)
- Concatenate it to links found so far
- Find the next "<a"

```
// Extract all the links from a web page
private String findLinks(String fullpage) {

    int tagPos,               // Start of <A tag specification
        tagEnd;               // Position of first ">" after tag start

    // A lower case only version of the page for searching
    String lowerpage = fullpage.toLowerCase();
```

```
      // Text of one A tag
      String tag;

      // The A tags found so far
      String links = "";

      // Paste stuff on end of page to ensure searches succeed
      fullpage = fullpage + " >";

      tagPos = lowerpage.indexOf("<a ",0);

      while (tagPos >= 0 ) {
         tagEnd = fullpage.indexOf(">",tagPos+1);

         tag = fullpage.substring(tagPos, tagEnd+1);

         links = links + tag + "\n";

         tagPos = lowerpage.indexOf("<a ", tagEnd);
      }

      return links;
   }
```

Miscellaneous interesting `String` methods

```
   public boolean startsWith(String s)
         // true only if this string starts with s

   public boolean endsWith(String s)
         // true only if this string ends with s

   public boolean equals(String s)
         // true only if this string has same sequence of chars as s

   public boolean equalsIgnoreCase(String s)
         // true only if this string has same sequence of chars as s
         // except capital & lower case letters considered the same

   public int lastIndexOf(String s)
   public int lastIndexOf(String s, int startIndex)
         // return index of last occurrence of s (occurring at or
```

```
      // before startIndex) in this string, and -1 if no match.

public String replace(char oldChar, char newChar)
      // Returns a new string resulting from replacing all
      // occurrences of oldChar in this string with newChar.

public String trim()
      // Returns a new string with leading and trailing spaces removed.

public int compareTo(String s)
      // Returns negative int if string before s in case-sensitive
      //    dictionary order;
      // returns 0 if equal
      // returns positive int if string after s in case-sensitive
      //        dictionary order.

public char charAt(int index)
        // Returns the character at the specified index.
```

Characters

Our `Strings` are made up of *characters*, so let's take a look at that idea. In some sense, a Java `String` is really an array of characters, but we don't treat it like a regular array. In C, strings are actually nothing more than an array of characters.

A character is what you probably expect roughly speaking, it is a keyboard character. This includes special function characters (like newline and tab), in addition to alphanumeric and punctuation characters.

A character is represented internally as a number an integer, in fact. There are various "universal" codes that can be used to represent characters:

- ASCII (256 chars) (American Standard Code for Information Interchange)
- EBCDIC (Extended Binary Coded Decimal Interchange Code)
- Unicode
  - Allows up to 65,536 chars.
  - About 35,000 currently in use.
  - First 256 are the same as ASCII codes (for compatibility).

To declare a variable to be of character type:

```
private char letter;
```

Use single quotes for character constants: `'H'` is a `char`; `"H"` is a `String`:

```
char letter = 'H';
```

Hangman

Let's look at the use of some `String` methods to play a simple game of "Hangman".

See Example: Hangman

This word/letter guessing game uses Java's `String`s but also reviews how we can read input from the keyboard in the terminal window and how to read input from a data file.

The first part of the program we want to look at is the construction and initialization of our list of words. These words are located in a file `dictionary.txt`, which contains about 127,000 words. We will put these into an `ArrayList` of `String`s.

The implementation makes use of Java's `Scanner` class, which allows us to read the contents of the file word by word, and add them to the word list.

The program then prepares to play the game. We create a Java standard random number generator (a `Random` object). Then we create another `Scanner` that allows us to read things typed in the terminal window.

The main loop involved picking a random word from the dictionary and presenting the player with a number of blanks equal to the number of letters in the word. The player is prompted for guesses (letters). Along the way, we keep track of which letters have been guessed and how many incorrect guesses have been made. If the player runs out of guesses, we have a hangman and the player loses. If the player has guessed all of the letters, the player wins. Otherwise, the word is displayed again, now with any guessed letters shown.