Highly Recommended (Optional) Programming Exercise (worth 0% of your final grade)

COMP-202B, Winter 2011, All Sections

March 1, 2011

Introduction

This exercise is a good way to study for the final exam (or even for the midterm). It's divided into many small methods, so you'll get good practice from it even if you don't complete the whole thing. Suggested strategy: try writing two methods a week for 7 weeks. If you're using this to study for a test, try writing some of the methods completely on paper (to simulate a test situation) before you write them on the computer.

The `String` class is one of the most useful classes among those provided by the Java Standard Class Library, but how does it actually work? **The purpose of this exercise is to allow you to gain some insight on how the internals of this important class can be implemented, while gaining experience writing classes that represent categories of objects. This exercise is also a very good review of 1D arrays and the `char` type.**

Finally, forgive the tone of the **MUST**s and **MAY**s; this exercise is actually from an old COMP-202 assignment for which the requirements were very strict. `:)`

Instructions

Write a class called `MiniString`; like `String` objects, objects that belong to the `MiniString` class represent ordered sequences of characters. Also like `String`s, valid character indices in a `MiniString` range from 0 (inclusive) to the number of characters in the `MiniString` (exclusive); that is, the i^{th} character of a `MiniString` has index $i - 1$.

`MiniString` objects **MUST** keep track of the character sequences they represent using a **PLAIN ARRAY** (of `char`); you **MUST NOT** use regular `String`s within the `MiniString` class under **ANY** circumstances, except if one of the method descriptions below explicitly allows the use of the `String` class. You **MUST NOT** use `ArrayList`s (or any other class which is part of the Java Collection Framework such as `LinkedList`) within the `MiniString` class either. Finally, you **MUST NOT** use the methods declared in the `Arrays` class to perform array processing.

Note that `MiniString` objects are immutable, meaning that the state of a `MiniString` object **MUST NOT** change once the object is created. This implies that, other than the constructors, the methods of the `MiniString` class **MUST NOT** change the state of the target `MiniString`, and no method or constructor should ever change the state of any parameter `MiniString`.

Also note that you **MUST** respect proper encapsulation practices; that is, the attributes and methods of your class **MUST** be declared using the proper visibility modifiers.

Your `MiniString` **MUST** provide the following `public` **INSTANCE** methods:

1. A constructor, which takes as its only parameter an array of `char`, and initializes the newly-created `MiniString` so that it represents the sequence of characters currently contained in the parameter array. The contents of the parameter array are copied, so that any subsequent modifications to the contents of the parameter array **MUST NOT** affect the newly-created `MiniString`.

   You **MAY** assume that the array received as parameter is not a `null` reference; in other words, your constructor does not have to handle the case where the array received as parameter is a `null` reference.

2. A constructor, which takes as its only parameter a `String`, and initializes the newly-created `MiniString` so that it represents the same sequence of characters as the parameter `String`.

   You **MAY** assume that the `String` received as parameter is not a `null` reference; in other words, your constructor does not have to handle the case where the `String` received as parameter is a `null` reference.

   You **MAY** use `String` objects and the methods declared in the `String` class to implement this constructor.

3. A method called `length()`, which takes no parameters and returns a value of type `int` representing the length of the target `MiniString`, that is, the number of characters it contains.

4. A method called `charAt()`, which takes as its only parameter a value of type `int`, and returns a value of type `char` representing the character in the target `MiniString` at the position given by the parameter. Remember that `MiniString` indexing works exactly like regular `String` indexing.

   You **MAY** assume that the value of the `int` parameter is a valid index in the target `MiniString`; in other words, you do not have to handle the case where the `int` parameter is negative, equal to the length of the target `MiniString`, or greater than the length of the target `MiniString`.

5. A method called `toString()`, which takes no parameters and returns a `String` representing the same sequence of characters as the target `MiniString`. You **MAY** use `String` objects and the methods declared in the `String` class to implement this method.

6. A method called `toCharArray()`, which takes no parameters and returns an array of `char` which contains the characters of the target `MiniString`. The length of the returned array **MUST** be equal to the length of the target `MiniString` as specified by a call to the `length()` method on the target `MiniString`. Furthermore, the character at position `i` in the returned array **MUST** be equal to the character which would be returned by a call to `charAt()` on the target `MiniString` with actual parameter `i`, for every value of `i` which is a valid index in the target `MiniString`.

   Subsequent changes to the contents of the array returned by this method **MUST NOT** change the state of the target `MiniString`.

7. A method called `equals()`, which takes as its only parameter a `MiniString`, and returns a value of type `boolean`. This method returns `true` if the target `MiniString` represents the same character sequence as the parameter `MiniString` (that is, both contain the same number of characters in the same order), `false` otherwise. If the `MiniString` received as parameter is a `null` reference, then the method **MUST** return `false`.

8. A method called `compareTo()`, which takes as its only parameter a `MiniString`, and returns a value of type `int`. This method compares the target `MiniString` to the parameter `MiniString` for order, and returns the following values:

   - A negative value if the target `MiniString` lexicographically precedes the parameter `MiniString`
   - A positive value if the target `MiniString` lexicographically follows the parameter `MiniString`
   - A value of `0` if the target `MiniString` and parameter `MiniString` are equal; this method returns `0` exactly when the `equals()` method described previously would return `true`.

The definition of lexicographic ordering is as follows:

> If two `MiniString`s are different, then either they have different characters at some index that is a valid index for both `MiniString`s, or their lengths are different, or both. If they have different characters at one or more index positions, let k be the smallest such position; then the `MiniString` whose character at position k has the smaller character code, as determined by using the `<` operator, lexicographically precedes the other `MiniString`. If there is no index position at which the target `MiniString` and the parameter `MiniString` differ, then the shorter `MiniString` lexicographically precedes the longer `MiniString`.

The above definition is adapted from the documentation for the `String` class found in the Java Standard Class Library.

You **MAY** assume that the `MiniString` received as parameter is not a `null` reference; in other words, your method does not have to handle the case where the `MiniString` received as parameter is a `null` reference.

9. A method called `concatenate()`, which takes as its only parameter a `MiniString`, and returns a new `MiniString` representing the concatenation of the target `MiniString` and the parameter `MiniString`, in that order. For example, if the target `MiniString` represents the character sequence `"COMP"`, and the parameter `MiniString` represents the character sequence `"-202"`, then the method should return a new `MiniString` representing the character sequence `"COMP-202"`.

   If the `MiniString` passed as parameter is a `null` reference, then the method returns a `MiniString` representing the same character sequence as the target `MiniString`.

10. A method called `substring()`, which takes as a parameter two values of type `int` called `start` and `end`, in this order. The method returns a new `MiniString` formed of the characters between position `start` (inclusive) and position `end` (**EXCLUSIVE**) in the target `MiniString`. Within the new `MiniString` returned by this method, the characters occur in the same order as in the original `MiniString`.

    You **MAY** assume that the value of `start` is greater than or equal to `0`, that the value of `end` is less than or equal to the length of the target `MiniString`, and that the value of `start` is less than or equal to the value of `end`; in other words, your method does not have to handle cases where the values of `start` and `end` do not satisfy these constraints.

11. A method called `substring()`, which takes as its only parameter a value of type `int`. This method returns a new `MiniString` formed of the characters between the position in the target `MiniString` specified by the `int` parameter, and the end of the target `MiniString`. Within the new `MiniString` returned by this method, the characters occur in the same order as in the original `MiniString`.

    You **MAY** assume that the value of the `int` parameter is a valid index in the target `MiniString`; in other words, you do not have to handle the case where the `int` parameter is negative, equal to the length of the target `MiniString`, or greater than the length of the target `MiniString`.

    Hint: Have this `substring()` method call the `substring()` method which takes two parameters of type `int`.

12. A method called `replace()`, which takes as parameters two values of type `char` called `oldChar` and `newChar`, in this order. The method returns a new `MiniString` representing the same character sequence as the target `MiniString`, except that every occurrence of `oldChar` is replaced by an occurrence of `newChar`. If `oldChar` does not occur in the target `MiniString`, then this method returns a `MiniString` representing the same character sequence as the target `MiniString`.

13. A method called `indexOf()`, which takes as parameters a value of type `char` followed by a value of type `int`, and returns a value of type `int`. The method searches all positions within the target `MiniString` between the one specified by the `int` parameter and the end of the target `MiniString`, and returns the smallest index in this range such that the character at this index in the target `MiniString` is equal to the `char` parameter. If the `char` parameter does not occur in this range, then the method returns `-1`.

Note that there is no restriction on the value of the `int` parameter. If it is negative, it has the same effect as if it were `0`; the entire `MiniString` will be searched. On the other hand, if it is greater than or equal to the length of the `MiniString`, the range will consist of no characters, so the method will return `-1`.

14. A method called `indexOf()`, which takes as a parameter a value of type `char`, and returns a value of type `int`. The method searches everywhere within the target `MiniString`, and returns the smallest index such that the character at this index in the target `MiniString` is equal to the parameter `char`. If the parameter `char` does not occur at all in the target `MiniString`, then the method returns `-1`.

    Hint: Have this `indexOf()` method call the `indexOf()` method which takes a parameter of type `char` and one of type `int`.