CS 101 - Algorithms & Programming I

Fall 2025 - Lab 5

Due: Week of November 3, 2025

*Remember the **honor code** for your programming assignments.*
*For all labs, your solutions must conform to the CS101 style **guidelines**!*
All data and results should be stored in variables (or constants where appropriate) with meaningful names.

The objective of this lab is to learn **static methods**. Remember that analyzing the problems and designing the solutions on a piece of paper *before* starting implementation/coding is always a best practice.

For the methods below, you should **not** use any built-in Java methods that perform the mentioned tasks directly (e.g., `String.toLowerCase()`, `String.reverse()`, `Character.isDigit()`, etc.). The mentioned methods **should be implemented from scratch, unless explicitly stated otherwise**. For this lab assignment, you can refer to the ASCII values and table here and here.

0. Setup Workspace

Start VSC and open the previously created workspace named `labs_ws`. Now, under the `labs` folder, create a new folder named `lab5`. In this lab, you are to have 2 Java classes/files (under `labs/lab5` folder) as described below. A third Java file containing the revision should go under this folder as well. We expect you to submit a total of 3 files, including the revision, **without compressing** them. Do *not* upload other/previous lab solutions in your submission.

First, we will design and implement some static methods. Then, these methods will be used in a program to perform various text operations based on user input.

1. Text Toolkit Program

In this part, you will implement a "toolkit" of static methods for performing various string analysis and manipulation tasks. You will also create helper methods to assist in these tasks.

Helper Methods

Before tackling the main problems, you will need to implement a few smaller, reusable helper methods. These will make your code for the larger methods cleaner and easier to write.

- `static char toLower(char ch)`
  This method should convert an uppercase letter to its lowercase equivalent. If the character is not an uppercase letter, it should be returned unchanged.

  Hint: In the ASCII table, the uppercase letters `'A'` through `'Z'` and the lowercase letters `'a'` through `'z'` are arranged in sequential blocks. The numerical difference between any uppercase letter and its corresponding lowercase letter is constant. You can find this difference by simply calculating `'a'-'A'`. By adding the difference to an uppercase character, you can convert it to lowercase.

- `static boolean isLetter(char ch)`
  This method checks if a given character is an alphabet letter (either uppercase from `'A'` to `'Z'` or lowercase from `'a'` to `'z'`). It should return `true` if it's a letter and `false` otherwise.

  Hint: You can again leverage the ASCII table, as the letters are arranged in the ASCII table in sequential order.

- `static boolean isDigit(char ch)`
  This method should return `true` if the character is a digit from `'0'` to `'9'`, and `false` otherwise.

  Hint: You can again leverage the ASCII table, as the digits are arranged in the ASCII table in sequential order.

- `static boolean isWhiteSpace(char ch)`
  This method checks for whitespace characters. It should return `true` if the character is a space (`' '`), a tab (`'\t'`), or a newline (`'\n'`). Otherwise, it should return `false`.

String Analysis

Implement the following methods to analyze string properties.

- `public static boolean isPalindrome(String str)`
  A **palindrome** is a word, phrase, or sequence that reads the same backward as forward. Your method should check if the input string is a palindrome. The check must be **case-insensitive** and should **ignore only whitespace characters**. For example, "Never odd or even" is a palindrome.

- `public static boolean areAnagrams(String str1, String str2)`
  Two strings are **anagrams** if they are written using the exact same letters in a different order. Your method should check if two input strings are anagrams. The check must be **case-insensitive**. For example, "Listen" and "Silent" are anagrams. Only take words into account while checking whether the two strings are anagrams for this task.

  Hint: A common way to check for anagrams is to count the occurrences of each character in both strings. If the counts for every character match, they are anagrams. You can do this by scanning the first string character by character, deleting an occurrence of that character from the second string, and checking if the second string becomes empty in the end. For this purpose, you may use Java's built-in `String.indexOf()` and `String.substring()`methods. Notice that you need first to create new versions of given strings to exclude characters that are not letters and perform the scanning operations in the subsequent stages.

- `public static int wordCount(String str)`
  The method should count the number of words in a given string. For this problem, a **word** is defined as any sequence of non-whitespace characters separated by white space. For example, the string "Hi there, CS101!" contains 3 words. Make sure to account for strings that don't contain any whitespace, such as single word inputs.

String Manipulation

Implement the following method to manipulate strings.

- `public static String slugify(String str)`
  A "slug" is a URL-friendly version of a string, often used in blog post URLs. This method should convert a string into a slug based on the following rules:
  1. Convert the entire string to lowercase.
  2. Replace any sequence of one or more whitespace characters with a single hyphen (`-`).
  3. Keep letters and digits. Remove all other characters (punctuation, symbols, etc.).

4. Ensure there are no leading or trailing hyphens.
    - For example, `"Hello, World! 2025"` should become `"hello-world-2025"`

    Hint: Appending characters to a `String` in a loop can be inefficient. For building new strings, it's better to use the `StringBuilder` class. You can append characters to a `StringBuilder` instance and then call its `ToString()` method at the end to get the final `String`.

Simple Menu for Text Toolkit

Create a new file under the `lab5` folder named `Lab05_Q1.java`. The `main` method in this class should provide a menu-driven interface that allows a user to select an operation from your toolkit, enter the required text, and view the result. The program should continue to show the menu until the user chooses to exit. You can access the sample run for this part of the lab assignment here.

2. Cipher Toolkit Program

In this part, you will implement several classic cipher and text manipulation algorithms as static methods.

Helper Methods

Just like in the first part, you will need a couple of helper methods for your cipher toolkit.

- `static boolean isLowerCase(char ch)`

This method should check if a character is a lowercase letter (from `'a'` to `'z'`).

- `static boolean isLetter(char ch)`

This is the same method as in Question 1. You can reuse it here. It checks if a character is a letter, either uppercase or lowercase.

Caesar Cipher

The Caesar cipher takes its name from Julius Caesar, who protected military messages by shifting letters three places, so A becomes D when encrypting and D returns to A when decrypting.

To explain it more, the Caesar cipher is a simple substitution cipher where each letter in the text is shifted a certain number of places down the alphabet. For example, with a shift of 3, `'B'` would be replaced by `'E'`, `'C'` would become `'F'`, and so on. The alphabet "wraps around," so `'Z'` shifted by 3 becomes `'C'`.

- `public static String caesarCipher(String text, int shift, boolean encode)`

    This method should take a `String`, an integer `shift` value, and a boolean `encode`. If `encode` is `true`, the text should be encrypted. If `false`, it should be decrypted. The cipher should affect both uppercase and lowercase letters but leave all other characters (numbers, punctuation, spaces) unchanged.

    Hint: Decrypting is the same as encrypting with a **negative shift**. The modulo (%) operator is essential for handling the "wrap-around" logic. Be careful when decrypting, as the result of a modulo operation in Java can be negative. If `(originalPosition - shift) % 26` is negative, you'll need to add 26 to get the correct positive position.

Atbash Cipher

The Atbash cipher is another substitution cipher where the alphabet is reversed. ‘A’ is swapped with ‘Z’, ‘B’ with ‘Y’, and so on. It was originally used in the Hebrew alphabet, but it can be applied to any alphabet.

- `public static String atbashCipher(String text)`

  This method implements the Atbash cipher. An interesting property of this cipher is that the same algorithm is used for both encryption and decryption. Your implementation should preserve the case of letters and leave non-alphabetic characters unchanged.

  Hint: For any given uppercase character `ch`, its Atbash equivalent is `(char)(‘A’ + (‘Z’ - ch))`. A similar formula applies to lowercase letters.

Text Reversal

This is a simple utility method for reversing a string.

- `public static String reverseText(String text)`

  This method should take a string as input and return a new string with all the characters in reverse order. For example, `”Programming 101! Should become ”!101 gnimmargorP”`.

  Hint: As with the `slugify` method, using a `StringBuilder` is an efficient way to build the reversed string.

Simple Menu for Cipher Operations

Create a new file named `Lab05_Q2.java` under the `lab5` folder. In this class, the `main` method should provide a menu-driven interface to test the cipher and reversal methods you implemented. The program should loop until the user chooses to exit. You can access the sample run for this part of the lab assignment here.