vim ~/.vimrc

PSA 1: Testers, Strings, and Debugging

Due: 11:59pm, Sunday 1/14/2018

Overview video (new): https://youtu.be/1RqbLi9fMTI

Overview and Getting Started with CSE 8B

In this first programming/problem solving assignment, the goal is warming up for this class and seeing how the course runs. This assignment introduces the basics of debugging and testing, while the major portion of the assignment, the programming portion, will focus on encrypting and decrypting. ***You should read all instructions, hints, and restrictions before copying any files and starting any work on the assignment.*** Manage your time wisely. Each part (including the README and adhering to style) may take different amounts of time. This assignment is out of 100 points with opportunities for 6 extra points.

We are here to help wherever we can, so the following documents are linked to tell you about the important aspects of the course. These are linked in every PSA. This PSA includes a small description for each document. To succeed in the course, you should understand how the course works, although some things may be more personally relevant than others. If there is confusion or contradiction, please ask for clarification on Piazza or in-person.

We recommend that you develop on the ieng6 servers or lab machines because the compiler and dependencies are set up there already. You may choose to develop locally, but we won't offer support there. Ensure that you use Oracle Java 8. Any issues that arise from using an incorrect version of Java cannot be regraded.

- Setting Up the PSA
  - These are the generic instructions for getting started with the PSA. You might need to refer to it any time you start the assignment. The starter files and how to get them are found in this document.
- Online Communication: Using Piazza, Opening Regrade Requests
  - We encourage students to reach out for clarifications and assistance, but under what scenario and situation would emailing a TA be appropriate? When is it not? Find out which sources of communication to get help or contact the staff, and when.
- Getting Help from Tutor, TA, and Professor's Hours
  - Assignments are meant for learning, which means they might be challenging. Read about getting help in-person, which may prove to save large amounts of time.
  - Lab and Office Hours (Always refer to this calendar before posting.)
- Academic Integrity: What You Can and Can't Do in CSE 8B
  - We expect all students to follow the honor of academic integrity, which exist mainly for fairness in your education.
- How to Use Vim and compile codes in terminal
  - CSE 8A used DrJava. However, in this class and future classes, you will be typing **much** larger amounts of code and Vim (gVim) lets you edit code with much more effectiveness and ease than DrJava. We will also use command line to compile codes.
- Style Guidelines
  - You are expected to follow the Style Guidelines in your final version of your programs.
- Submitting on Vocareum and Grading Guidelines

- When you finish your programs, how do you submit them? What do we generally look for when grading? Find out here.

Table of Contents

Part 0: Research and Academic Integrity Agreements

Completion of this section is required for your assignments to be graded.

1. You MUST read and electronically sign the CSE 8B Integrity of scholarship agreement before going any further. Integrity of scholarship is a critical part of this course. Failing to do so will result in grades of 0 until you sign the agreement. If you have any questions or are unsure about how any part of this agreement applies to CSE 8B, please ASK your instructor. Most cases of academic dishonesty occur because a student had not taken the time to understand the policy. Make sure you complete the academic integrity tutorial before signing the agreement.
   Link to electronically sign the Integrity of Scholarship agreement

2. The next part is for stating whether or not you agree to participate in a research study we are conducting as part of this class. You won't have to do anything special to participate, and your data will remain completely confidential. More information can be found on the consent form linked below. Link to informed consent form. You will receive 2 points for this PSA if you complete this consent form whether you agree to be part of the study or not.

Part 1: Starting with Linux; Vocabulary [2 Extra Credit Points]

We assume you will use Linux (CentOS as found in the labs). macOS will be similar, but not exactly the same. If you have Windows 10, you may use the lab computers or install the virtual Ubuntu Bash inside Windows 10. See https://docs.microsoft.com/en-us/windows/wsl/install-win10 . Regardless of your operating system, we highly recommend using the lab machines in B220-260 or connecting to them through SSH. Make sure your code compiles on the lab machine as well.

In this section, you will start practicing editing files using the command line found in the Terminal.

The terminal is much like a file explorer, but more powerful since it is a command line where you perform tasks. The most important commands for CSE 8B will probably be **`ls, pwd, cd, vim, javac, and java.`**

- **`ls - show files and folders in current directory`**
- **`pwd - show the path of current directory`**
- **`cd - change directory. no argument (meaning `cd`) will return you to the default directory. cd with argument (`cd pa1`) will look for pa1 in current directory and enter. `..` means up one directory, so `cd ..` means go back up one directory.`**
- **`vim filename - opens a file called filename in the Vim text editor.`**
- **`javac AFile.java - compiles this java file using the java compiler. If you need to include other files in the compilation process, sometimes javac *.java may be helpful.`**
- **`java AFile - executes the compiled java file. Note that it is not `java AFile.java` nor `java AFile.class`.`**
- **`for more, see https://files.fosswire.com/2007/08/fwunixref.pdf`**

WARNING: **`rm`** means delete. The -r option removes the directory and its contents recursively and the -f option means force. This means that `**rm -rf**` will FORCE THE RECURSIVE REMOVAL OF EVERYTHING IN THE DIRECTORY and `**rm ***` means remove all files in the directory. Use with extreme caution. If you accidentally delete files on the CSE lab machines, we might or might not have backups -- open a private Piazza post as soon as possible if this is the case.

Remember to backup your code often. You can do this by making a copy of your folder, rename it, and save it somewhere safe and private. To copy a folder and its contents, do `` `cp -r originalFolder originalFolderBackup` `` which will make a copy of the original folder into a folder named originalFolderBackup.

You should have already copied all the starter code from the server located at `~/../public/psa1`. If you haven't yet, refer back to Setting Up the PSA for more details on how to obtain the starter code. Navigate your Terminal into the folder with your PSA1 contents. Then open a new file using vim: `` `vim README.md` ``.

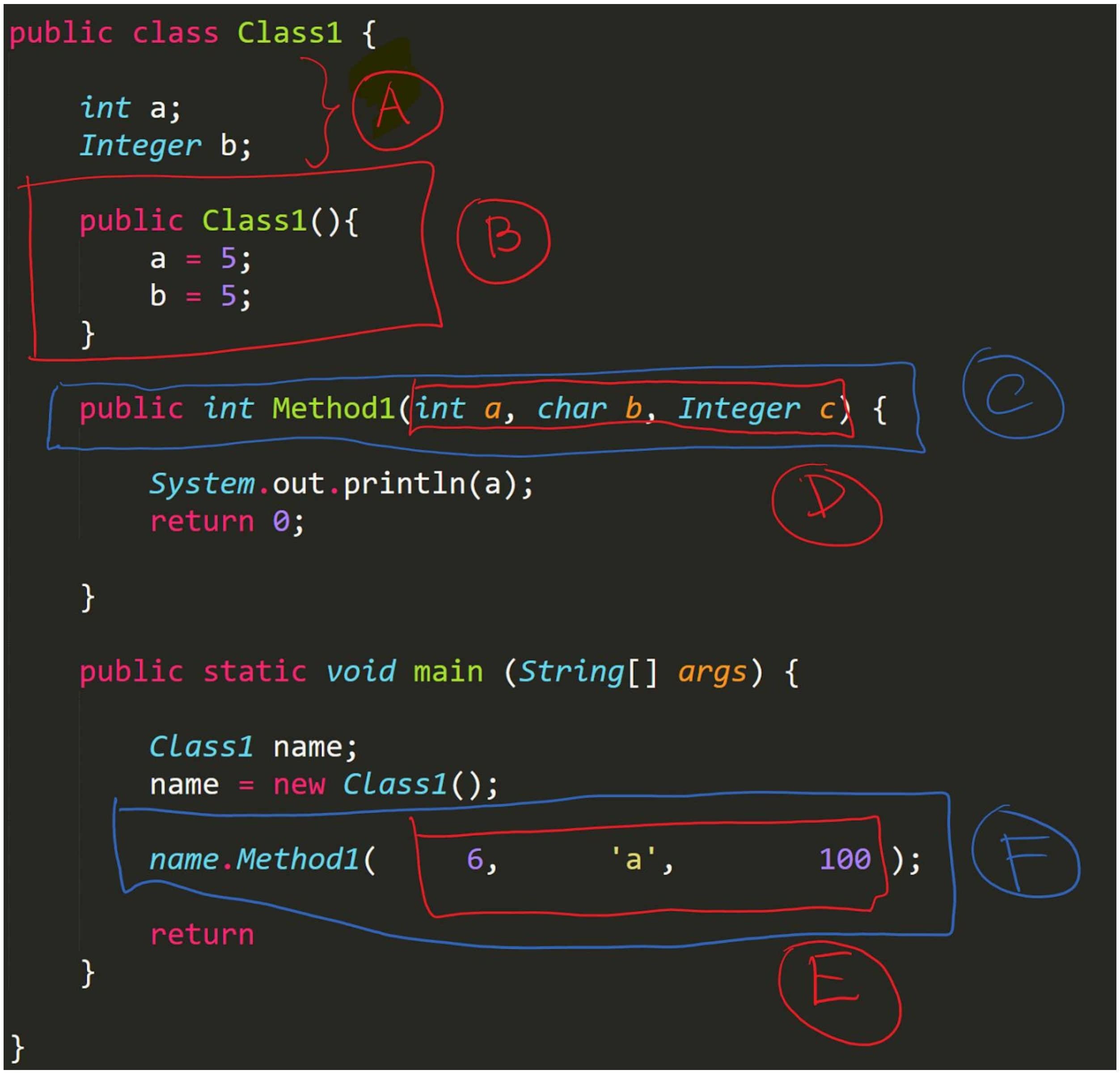
This takes you to a different interface in the Terminal. You will initially be in "command mode". Go to edit mode by pressing i. Type some stuff to test it out. Exit it by pressing Esc. Save by giving the Vim command :w. Exit by the Vim command :q. Save and quit by doing :wq or :x. It will take you back to the command line.

Use the `ls` command to see if the file exists, and use `cat` to display its contents: `` `cat README.md` ``.

Try the online Vim tutorial: http://www.openvim.com/tutorial.html

Here is your task. Vocabulary: Write the following in a file named "`README.md`" in a section labelled `Part 1`:

- For Sections A through F, describe what the boxed items are, using the information from `terms.java`.
- Explain why C and F are matched up and why Java is able to identify and connect the two. (Hint, use `terms.java` and notice the color-coded letters and boxes)



```
public class Class1 {

    int a;
    Integer b;

    public Class1(){
        a = 5;
        b = 5;
    }

    public int Method1(int a, char b, Integer c) {

        System.out.println(a);
        return 0;

    }

    public static void main (String[] args) {

        Class1 name;
        name = new Class1();

        name.Method1(    6,        'a',        100 );

        return
    }

}
```

Submit early and often: When you have finished with the section, submit this section's Files to turn-in and all previous Files to turn-in together onto Vocareum. Instructions are linked in the beginning.

Files to turn-in: README.md

Part 2: Debugging Code [30 Points]

The two files you need for this part:

FunWithIntArrays.java: This class is designed to have some useful features for int arrays but is marred with errors (compile, runtime and logic errors). There are a number of methods in the class whose expected behavior is commented.

ArraysTester.java: This code tests the class FunWithIntArrays. This file is to help you test your changes in the FunWithIntArrays class. You do not have to upload this file on Vocareum when turning in your work.

First, try to compile the two files. You can do this with the `javac` command, where "c" stands for compiler. The following command will compile `FunWithIntArrays.java` in the bash terminal:

```
javac FunWithIntArrays.java
```

You will notice that several errors will occur and that the class will not be compiled. **Your task is to edit the `FunWithIntArrays.java` file in order to fix these errors.** You can open it in vim with the following command:

```
vim FunWithIntArrays.java
```

When you open the file, you will be in **command mode**. This is for quickly navigating and manipulating parts of the file. Press the `i` key, which will bring you into **insert mode**. This is closer to a general text editor, in which you can navigate with the arrow keys and insert text with keyboard. Once you have made the necessary changes, you can exit insert mode and return to command mode with the `escape` key. Once you are in command mode, exit the editor by typing the following: `:wq` (notice that this includes the colon at the beginning). This will write any changes to the file and quit vim. For an overview of vim and the terminal, check out the vim and compiling code tutorial discussion slides. You can learn much more about how to use vim from the Useful Links section of the CSE 8B homepage.

When you believe you have fixed the errors in the program, you can attempt to compile again with the javac command. If it compiles successfully, it will generate a file named **FunWithIntArrays.class**. This is the compiled version of FunWithIntArrays.java. Once this works, compile ArraysTester.java with javac. Then you can run the program with the following command:

java ArraysTester

Note that this only works because the ArraysTester class has a main method defined in it. Make sure not to include the file extension when running this command.

1. Part of this assignment is to start learning the Vim editor. Again, an overview of vim and compiling code is given in the discussion slides. Several additional resources on Vim are under the Useful Links section of the CSE 8B homepage. Check them out.
2. After debugging and fixing the FunWithIntArrays class, compile the ArrayTester.java test driver (original file).
3. Run the ArrayTester program. The output should be the following:

```
Creating Initial Array:
7, 4, 1, 8, 12, 32, 64, 13,

Creating Array Copy:
```

```
7, 4, 1, 8, 12, 32, 64, 13,

Min element is: 1

Max element is: 64

Average value is: 17.625

Testing Sorted Array
1, 4, 7, 8, 12, 13, 32, 64,
```

List the bugs found in the **file header** of **`FunWithIntArrays.java`** and include a concise and clear description of why the bugs were wrong. Do not just list bugs you encountered in the assignment; your descriptions must detail the issues about the bug. They must be specific to **`FunWithIntArrays.java`**.

For example, if you fixed the for loop below:

```
for (int i = array.length; i >= 0; i--)
```

to avoid going out of bounds, your entry would look like:

```
Runtime Error - Array out of Bounds

Incorrect: for (int i = array.length; i >= 0; i--)

Fix: for (int i = array.length - 1; i >= 0; i--)

Explanation: Correction starts at the first element which is at index array.length-1
      rather than array.length.

// (Extra spacing)

<Next Error>
```

Put a newline between each error for readability. See above box where it says "(Extra spacing)".

Submit early and often: When you have finished with the section, submit this section's Files to turn-in and all previous Files to turn-in together onto Vocareum. Instructions are linked in the beginning.

Files to turn-in: FunWithIntArrays.java (Include README.md with your submission!)

Part 3: Testing Code [10 Points]

What is testing code? Here, we will be exploring situations where you will already have some pre-existing compilable code, but you now need to confirm the CORRECTNESS of that code. You might have pre-existing code from starter code, or from methods you wrote yourself. Correctness means the correct behavior of a given program. Programs have methods, so method correctness is this: Given some input arguments, when the program runs its method on those input arguments, is the output behavior correct? If the program crashed (an ERROR is thrown), is that crash correct? If an output is given to us, is this output correct?

Consider the following scenarios:
Suppose I have a method that is given a list of words as an array of Strings, a target word as a String, and the method is to algorithmically compute how many words in the array are anagrams of the target word. Here are the following cases:

PROGRAM CRASH
Input: {"abc", "ab", "c"} and "ba"
Output: *Indeterminant due to ERROR.*

Some error message is printed to terminal like:
Exception thrown. These files at these lines are causing the issue.

In this case, the behavior is completely wrong and just plain bad. First, none of the inputs were null. The array did not contain any null references. None of the strings were empty so this was not an edge case. This is an absolutely normal case. The program should not crash for this.

WRONG OUTPUT
Input: {"abc", "ab", "c"} and "ba"
Output: Two anagrams matched. They are "ab" and "c".

Okay, in this case, at least the program did not crash. However, since the code successfully compiled and run, we now have to look at why the output is incorrect. This is slightly harder to debug and requires you to go back into the code and test it. Oftentimes, System.out.println is helpful here. This type of error is a calculation, a logic, a human error. Even if the programmer thinks the code he wrote is right, if the code outputs the wrong result, then the code is wrong.

In this case, the incorrect behavior is that the wrong behavior is given back. The code must be fixed.

Good test inputs include the following:

- int
  - Has the smallest possible value (MIN_VALUE). This is an edge case.
  - Has the largest possible value (MAX_VALUE). This is an edge case.
- String
  - Is an empty String: "" This is an edge case. (Do not copy and paste this.)
- Objects in general
  - Is a null reference. This is a null case.
- arrays in general
  - Is a null reference. This is a null case.
  - Is a non-null reference, but the array contains a null reference. This is a null case.

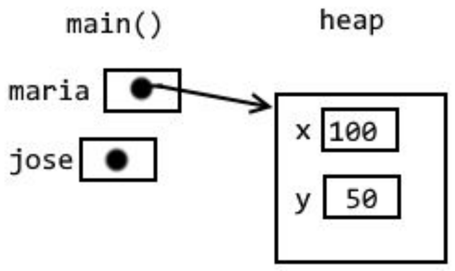
    - Has a length of 0

For other types and Classes, think about what kind of edge cases are possible.

Note about null references: References are variables that point to the location of an object in the heap. Internally, they are the memory address at which an object can be found. A **null reference** is reference which explicitly does not point to any object. This is the "zero-like value" of any reference. If we executed the following code (in main):

```
Turtle maria = new Turtle(100, 50);
Turtle jose = null;
```

Our memory model would be the following. The reference jose does not point to any object in particular, which is why it would not make sense to make a call such as `jose.forward(10);` This will result in a NullPointerException which will cause the code to fail.



EXCEPTIONS
Here's a small snippet from an excellent shorthand guide quoted from a tutor for identifying errors:

- ArrayIndexOutOfBoundsException/IndexOutOfBoundsException
    - Will be accompanied by the index that the program was attempting to access, and sometimes the actual size of the data structure.
- NullPointerException
    - This usually means an attempt was made to call a method on a null value or modify a reference to a null value. It might mean an object was not initialized.
- NoClassDefFoundError
    - The class file for this java Class was not found. The java file might not have been compiled.
- OutOfMemoryError
    - A data structure in the program is being increased in size indefinitely.
- StackOverFlowError
    - Infinite recursion.
- InputMismatchException
    - The type expected is not the type that was given. Example: Scanner, when it tries to read in the next integer with nextInt() but ends up reading letters.
- ArithmeticException
    - Mathematical error. Example: tried to divide by zero.

LOGIC MISTAKES
These types of errors will generally happen when your file is (1) compilable and (2) runnable. If you passed these two basic checks, then you are on your way to working on the core of your program: ensuring functional correctness.

Here is an example. I define a method int add(int first, int second) that will take in two ints and return the sum as the type int. If I did add (5, 2) and was returned 8, then clearly the code compiled and ran. However, the result is wrong. Here is the catch: ***computers do exactly what you tell it to do*****.** Because of this, computers cannot know what is right or wrong. Even with machine learning, computers are still doing exactly what you tell them to do. (They merely run learning algorithms). Therefore, ensuring your own logical correctness is pivotal to succeeding in this course. In the provided methods, all the programs will compile. But while some have exceptions, others will have logic errors. It is up to you to test the methods and determine whether the method has a logic error. You can do this by supplying the method a series of inputs and check the outputs. If the output is the incorrect value, then you know you have found a logic error.

On exceptions versus logic mistakes, we love exceptions. Java always tells us where the exception occurred by providing all implicated Java files and the lines that they were on. Take advantage of the Java error messages. They are invaluable for your debugging - for this assignment, they are necessary for your testing. However, debugging logic errors will take up the vast majority of your time. Compile errors and exceptions are easy to fix, but personal logic mistakes are harder and may take several more hours to fix. This is why we strongly recommend starting early on your assignments, ALWAYS. Be sure to plan out on paper what code you will write. Draw memory model diagrams. And finally, testing and debugging is the process by which you will succeed in your programming assignments and, therefore, CSE 8B. Learn it.

Here is your task. For this portion of your assignment, you will NOT be rewriting code. You will NOT be debugging code. In fact, the code will be completely hidden from you. In this assignment, you will identify wrong behavior. Wrong behavior can be one of the two possible choices as elaborated above. We give you a method that is defined 8 times inside `AnagramFinder.class`. Each of these methods will contain a DIFFERENT wrong behavior. It is up to you to do method calls on these methods and determine what methods have what wrong behavior.

The AnagramFinder class has 8 methods of interest: `p1`, `p2`, `p3`, `p4`, `p5`, `p6`, `p7`, and `p8`. Each of these methods has the same parameters and return type except their name is different. Each of them was intended to perform the same task: given an array of strings `s` and one string `t`, find all strings in `s` are anagrams (have the exact same characters, but in a different order). The method `p1` is the "correct" implementation of this method. The methods `p2`, `p3`, `p4`, `p5`, `p6`, and `p7` each have a single mistake or issue when given certain inputs and the method `p8` has more than one issue, but you only need to identify one wrong behavior to earn credit.

Getting started. The method `p2` is almost correct, but the tutor who wrote it forgot to check whether the second parameter, `t`, was null. Write a tester method named `testP2` in `AnagramTester.java` which demonstrates that the method's behavior is incorrect.

Similarly, the method `p3` is very close to the correct behavior, but only checks that all the character in `t` appear the correct number of times in the elements in `s` (that is, if `t` is "abc", it will look for strings that contain exactly 1 a, 1 b, and 1 c, but looks at no other criteria). Write a method named `testP3` that demonstrates this mistake. You will have to investigate the behavior of `p4`, `p5`, `p6`, `p7`, and `p8` **on your own** and find out what inputs will cause them to have incorrect behaviors, then write tester methods for each one that demonstrates these issues.

If you are looking for something more to do in this assignment, you can revisit `p8`. There are in fact several mistakes in this method. See if you can find them all. **Note:** for the purpose of this section, two strings that are the exact same are not considered to be anagrams. The true definition is much more nuanced, but this is not one of the errors.

When you find a mistake in one of the methods, you should document it in **`README.md`** in a section labelled **`Part 3`**. Include at least 1) a summary of the error 2) an example input that will make it fail, and 3) the type of error (program crash or wrong output). Consider also including 4) a hypothesis about how the method was written and why this error occurs and 5) how critical or severe the mistake is.

A statement on integrity. It is very easy to cheat in this portion of the assignment. But the only one who loses will be the cheater. This portion of the assignment is designed to teach students lessons that will prove deeply valuable throughout the CSE major, and throughout entire careers. Only the students who follow the instructions in this assignment with strong integrity and honesty will be able to learn the techniques required.

If you are finding that you are spending more than two hours on this, you may want to ask for help. As stated before, we are here to help. Post on Piazza with your questions. Do not post it privately unless you are posting your own code. For tougher questions or deeper confusions, come to tutor hours.

Files to turn-in: AnagramTester.java, README.md. (Also include FunWithIntArrays.java in your submission!)

Part 4: Caesar Cipher [50 Points]

Background

A *cipher* is an algorithm that takes a message in the form of a sequence of characters called the "plaintext" and returns another sequence of characters called the "ciphertext" which is

1) apparently hard to read, but
2) can be converted back into the original plaintext message. Turning the plaintext into the ciphertext is called encryption; reversing the process is called decryption.

A *substitutional cipher* is an algorithm that creates a ciphertext by simply substituting, for every occurrence of a particular letter of the alphabet in the plaintext, another particular letter of the alphabet; recovering the plaintext then just involves doing the reverse substitution. In this part, you will implement a kind of substitutional cipher called a *rotational cipher* or *Caesar cipher*.

Substitutional ciphers such as the Caesar cipher have been used historically to encode secret diplomatic and military messages, but these days they are not considered very good ways to hide information that you really want to keep secret. In fact, you could fairly trivially write a method that can break one using just character frequencies. More sophisticated modern techniques exist and should be used if you want strong security. But still, substitutional ciphers are useful in some contexts; for example, the "rot 13" rotational cipher is sometimes used to obscure (usually controversial or potentially offensive) text in email or newsgroups.

Functionality

The rotational cipher you will implement in this assignment will affects only *letters* in the plaintext; numbers, punctuation, etc. are not changed. Here's why it is called a rotational cipher: Imagine the letters A,B,C,...,X,Y,Z written in order around the rims of two wheels on the same axle, and the wheels lined up so that A on one wheel is next to A on the other, B is next to B, etc. If you peeled the letters off the rims of the wheels and laid them out side-by-side, it would look like:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Now rotate one of the wheels by one position, and you have the correspondences for the "rot 1" cipher, that substitutes B for A, C for B, ... , and A for Z. Again, laid out side-by-side, it would look like:

```
BCDEFGHIJKLMNOPQRSTUVWXYZA
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

If instead you rotated one position in the other direction, you have the "rot -1" cipher, which is equivalent to "rot 25". Laid out, it would look like:

```
ZABCDEFGHIJKLMNOPQRSTUVWXY
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The example shown is a mapping that each letter in the input string should follow. This means that each letter in the input string would have an offset applied to it, mapping into the letter corresponding to the mapping scheme defined by the rotation value.

NOTE: The example of the two wheels spinning shows the mapping scheme of each letter. It does not mean that we are rotating the string as a whole.

"Rot N" ciphers are possible for any integer value of the offset N, but many of them are equivalent; there are only 26 distinct rotational ciphers (including the "rot 0" 'identity' cipher shown first above that substitutes each letter with itself, changing nothing). In particular, note that "rot 13" is equivalent to "rot -13", and so it decodes itself:

NOPQRSTUVWXYZABCDEFGHIJKLM
ABCDEFGHIJKLMNOPQRSTUVWXYZ

An important detail: In the rotational cipher you will implement, upper-case letters are always substituted for upper-case letters, and lower-case ones for lower-case ones. The amount of rotation is the same for each. So, you could think of a rotational cipher as really having two sets of wheels, one pair for upper-case and the other for lower-case letters, with each set of wheels rotated the same amount; for example for "rot 13" you would have the correspondences

NOPQRSTUVWXYZABCDEFGHIJKLM
ABCDEFGHIJKLMNOPQRSTUVWXYZ

nopqrstuvwxyzabcdefghijklm
abcdefghijklmnopqrstuvwxyz

Another example:

```
Plaintext:  THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
Ciphertext: QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD
```

In this example, deciphering is done in reverse, with a right shift of 3.

You will use two ways to implement the cipher: (a) by using the standard library Java class String, and (b) by using the StringBuilder class. For both parts, you would be using the Character class provided in the **standard Java library**.

Part 4A: [20 of the 50 Points]

Strings are made of Characters. You can get a specific Character based on the index of the String. So if you iterate through the String, you can consider each Character individually. Then perform the change on the Character. Create a new empty String, and then for each encrypted/decrypted Character you get back, concatenate it to the String using the + symbol. For example: exampleString = exampleString + newCharacter;

To implement this part of the cipher, complete the following methods to your Caesar class (each of which is described in more detail below):

- **`public static String encrypt(String s, int rotation)`**
- **`public static String decrypt(String s, int rotation)`**
- **`private static char letterOperation(char letter, int rotation)`**

Here are the details for each method:i

`public static String encrypt(String s, int rotation)`

This method takes a **String** and an **int** and encrypts the String by rotating each character which is an upper or lower case letter by the amount rotation. It should **not change non-alphabetic characters.** This method returns the encrypted **String**.

`public static String decrypt(String s, int rotation)`

This method will decrypt a **String** by applying the **int** rotation *in the opposite direction*. So a string encrypted with one rotation should should be recovered by calling decrypt *with the same value for rotation*. This method returns the decrypted **String**.

`private static char letterOperation(char letter, int rotation)`

This method will take a character(char) as an input and perform rotation (indicated by the int parameter) only if the character is an alphabet(Uppercase or Lowercase) and return the rotated alphabet. If the character is not an alphabet, it will return the original character. This method should be called when encrypt and/or decrypt methods are used. Please note that this method is private.

- For this PSA, you are **NOT** allowed to use the built-in **toCharArray** method provided by standard Java library, nor the other encryptTwo/decryptTwo method you will write. The penalty for doing so is a zero for this portion of the assignment.
- **Note: You cannot change the signature (name, return type, parameters) of any method above. The signatures must remain unchanged otherwise our testers will fail and assign 0 points.**

Submit early and often: When you have finished with the section, submit this section's Files to turn-in and all previous Files to turn-in together onto Vocareum. Instructions are linked in the beginning.

Files to turn-in: Caesar.java (Include previous parts' files! FunWithIntArrays.java, AnagramTester.java, README.md).

Part 4B [20 of the 50 Points]

For this part, you are required to use the StringBuilder class, which is a class similar to String class and is also included in the java.lang package. The intention of this section is for you to implement a behavior using a class that has already been written. Refer to the official documentation for StringBuilder for all information regarding StringBuilder itself. (PDF Version in case link doesn't work)

By using StringBuilder objects, we would be able to manipulate sequences of characters much faster runtime than if we used String objects. The reason is that whenever we manipulate a String object, we create new objects, which takes up more memory. On the other hand, since we are able to manipulate a StringBuilder object within itself, the runtime of it would be faster.

To implement this part of the cipher, complete the following methods to your Caesar class (each of which is described in more detail below):

- **`public static String encryptTwo(String s, int rotation)`**
- **`public static String decryptTwo(String s, int rotation)`**

While the parameter types, the return type, and the behavior of these methods are identical to Part A's methods, the implementation details (the algorithm) will differ.

You will initiate an instance of a StringBuilder and use it to manipulate the string in progress. The StringBuilder keeps the String inside (StringBuilder has a String inside it) and manipulates that string. Use method calls to instruct how you want to manipulate the string.

When the string is finished, make the StringBuilder return the String to you, and then return the String.

- For this PSA, you are **NOT** allowed to use the built-in **setCharAt** method provided by standard Java library, nor the other encrypt/decrypt method you will write. The penalty for doing so is a zero for this portion of the assignment.
- **Note: You cannot change the signature (name, return type, parameters) of any method above. The signatures must remain unchanged otherwise our testers will fail and assign 0 points.**

Submit early and often: When you have finished with the section, submit this section's Files to turn-in and all previous Files to turn-in together onto Vocareum. Instructions are linked in the beginning.

Files to turn-in: Caesar.java (Include previous parts' files! FunWithIntArrays.java, AnagramTester.java, README.md).

Part 4C [10 of the 50 Points]

Finally, we want to see how much more efficient using StringBuilder is compared to just using String and Character's methods.

We have provided a file for you called "script" that tests the runtime of the two method you wrote on encrypting. To run it, type the following Linux commands:

```
chmod u+x script
./script
```

The script will show you the time of running encrypt as implemented using StringBuilder, and encrypt as implemented using String concatenation. An example of it is:

```
Timing for encryptTwo method:
real   0m0.111s
user   0m0.222s
sys    0m0.333s
Timing for encrypt method:
real   0m9.954s
user   0m9.999s
sys    0m10.001s
```

NOTE: This is an example. The only thing you need to get "correct" is to show the output similar to above. encryptTwo should be faster than encrypt, but your methods do not need to match these times. We only care about correctness for now. No need to worry about efficiency.

By using the script, answer the following questions in **`README.md`** in a section called **`Part 4`** :

1. Run the script three times. Write down the "real" runtime for each method of each trial in the file.
2. We provided a reason or two why StringBuilder should be more efficient. Explain the reasoning in your own words.
3. Compare and contrast how easy/hard, fast/slow, straightforward/complicated it was to code Part A in contrast to Part B.

Another thing, for **2 Extra Credit Points**, also submit your **CaesarTester.java** containing test cases that you wrote. It must be documented and styled well, including in the method headers what cases you tested. As long as your file convincingly proves to the grader that you indeed wrote test cases for Caesar, you will get these points.

Submit early and often: When you have finished with the section, submit this section's Files to turn-in and all previous Files to turn-in together onto Vocareum. Instructions are linked in the beginning.

Files to turn-in: README.md (Include previous parts' files! FunWithIntArrays.java, AnagramTester.java, Caesar.java)

General Hints and Other Remarks for Parts 1-4

1. ***START EARLY! Go to discussion sections and tutoring hours for help.***

2. Before you write any Java code, be sure you clearly understand the concept of a rotational cipher. Here are some examples of input-output behavior of the encrypt and decrypt methods: The rotational offset can be any int (positive, negative or 0); but any rotational offset will be equivalent to one in the range 0 through 25. For example, it can be seen that a rotation of -39 is equivalent to one of 13, and a rotation of 26 is equivalent to one of 0.

Thinking about this in terms of what the "edges" are might help you here.

3. The standard library Java class Character contains some static methods that may be useful to you in this assignment:

```
public static boolean isLetter(char c)
public static boolean isUpperCase(char c)
public static boolean isLowerCase(char c)
```

The String class defines many instance methods, and so these are instance methods that every String object has. Some important ones are described in Chapter 4 of the text. For this assignment, the most useful ones are probably these:

```
public int length()
public char charAt(int index)
```

4. Note that in Java, Strings are immutable, that is, no String instance method can change anything about the contents of a String object once it is created. In particular, the String object passed in as argument to your **`encrypt(String, int)`** and **`decrypt(String, int)`** methods cannot itself be modified by those methods. Instead, you will loop through the chars in that String, translating them one-by-one according to the required rotational cipher, and building up new Strings by string concatenation, until you get the resulting String you want, and then return that String.

5. One thing that can help in correctly computing the rotational substitution for a letter is understanding how to deal with char values as numerical values. Given the numerical value of a char, you can add a number to it (an offset, given by the rotation), and so get the char for the corresponding ciphertext letter.

6. To operate on the numerical value of a char, it can be cast to int, or assigned it to an int variable, or in some contexts it will be automatically converted to int for you. The value so obtained is the Unicode code for the char, which for us is the same as the ASCII code. ASCII code charts are available many places online; looking at such a chart, you can see that all the uppercase letters are in order, and have int values in the range 65 (which is the int value of the char 'A') through 90 (which is the int value of 'Z'). Similarly, the lowercase letters have the 26 int values in order in the range 97 through 122, which are the int values of 'a' and 'z' respectively.

7. When computing a letter substitution, the resulting letter has to have an int value in the appropriate range: uppercase letters go only to uppercase letters, and lowercase to lowercase. So, when applying an offset to a letter, you have to make sure you "wrap back around" so you do not exceed the limits of the range. This "wrapping around" makes sure the transformation is a rotation, and not just a simple offset.

As an example, suppose you want to compute the substitution for char 'd', with a rotation of -40. This is equivalent to a positive rotation of 12. (Make sure you handle any rotation amount, including negative rotations, correctly!) Adding this offset to the int value of 'd', we get

```
(int)'d' + 12
```

as the numerical value of the letter we want. We can convert back to char with a cast:

```
(char) ((int)'d' + 12)
```

which is the char 'p'. Now suppose you want to apply the same rotation of 12 to the char 'Q'. The expression

```
(char) ((int)'Q' + 12)
```

yields an int value corresponding to character ']'; but it should be 'C'. The problem is that adding 12 to 'Q' gives an int greater than 90, so it goes past the end of the range of the uppercase chars. If you take this approach, you need to figure out how to "wrap back around" in all cases like this.

8. Different from String objects, a StringBuilder object is mutable. We are able to change the calling object itself. The class contains methods such as append and insert that would enable us to alter the objects themselves. This means that while it's not possible to "edit" a String, it's possible to "edit" a StringBuilder.

Example:

To create a StringBuilder object of the string "testin":

```
String name = "testin";
StringBuilder stringObj = new StringBuilder(name);
```

To append the char 'g' to the end of stringObj:

```
stringObj.append('g');
```

At the end,

```
System.out.println(stringObj);
```

would print:

```
testing
```

For more methods that could be used for the StringBuilder Class, refer to:

https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html

9. The actual coding of the assignment should not take a long time. However, for you to do it efficiently, you should plan out your algorithm and brainstorm what you intend to code. Don't just start coding, or this PSA may take more time to complete than reasonable.

10. Be sure to read all provided instructions in the writeup, all comments provided in the code, and all questions and notes posted by students and staff on Piazza. This assignment is reading-intensive, but when you understand what is happening, the amount of time you spend coding should be reasonable. The methods you write (in particular part 4) can be written in a way that one method builds on top of another.

README.md [5 Points]

You should at this point have 3 sections in "`README.md`". In another section labelled **`Summary`**, write a concise summary for each part. Explain what were the requirements of each part, what you learned, and what your programs, testers, or both did. Your descriptions should be concise and understandable.

The audience of the summary is anyone who does not know any programming or computer science at all. As a rule of thumb, write as if you were writing for an impatient 5-year old boy or a non-computer scientist grandmother. This means use absolutely **no** Java or CSE terms like "methods" and "variables". High-level terms like "program" are fine. Describe the general intention and procedure, as well as your methodology.

Since there are four parts, there are four sections in your summary. Each section includes the brief summary of each part, the description of the programs and testers, and what you learned. Therefore, you will have four parts, each with the summary, descriptions, and explanations of what you learned. For example, for the tester section, a summary could be:

- **`"In this section, we wrote testers for code we were not able to read. Rather, the testers were based on the documentation and our task was to figure out whether the functions had correct behavior."`**
- **`"Tester.java contains the code I wrote to test the provided functions for correctness."`**
- **`"I learned it's possible for a function to be mostly right, but can be wrong in rare cases."`**

Do write the README in your own words, though.

Style [5 Points]

Please refer to the style guidelines.

Submitting the Assignment

How to Submit on Vocareum

Submission Files: (Make sure your assignment submission contains all of the files and they work on the ieng6 lab machines!)

- **FunWithIntArrays.java**
- **Caesar.java**
- **CaesarTester.java (Optional)**
- **AnagramTester.java**
- **README.md**

Maximum Score Possible: 106 out of 100 Points

If you noticed after each part, "Submit early and often" was written. Submitting your code after each section is one way of doing it. As you progress through this assignment, we reminded you to submit at least part of the assignment until the entire assignment is sent. In future PSAs, it is up to you to remember to submit early and often.Vocareum allows you to submit your code multiple times, and we will grade the latest submission by default.