University of Aberdeen
School of Natural and Computing Sciences
2025 – 2026

Programming assignment – Individually Assessed (No Teamwork)

Title: JC2002 – Java Programming Coursework	This assignment accounts for 30% of your final mark in the JC2002 course.

<u>Learning Outcomes</u>

On successful completion of this assignment, the student will prove to be able to:

- Write and run basic Java applications.
- Evaluate how to structure a Java application.
- Implement functional and executable code for a Java application.
- Analyse simple problems and solve them with a Java computer program.
- Apply object-oriented programming concepts for creating components in a Java program.

Information about plagiarism and collusion: The source code and your report may be checked for plagiarism. Please refer to the documentation on the Academic Integrity section of the University of Aberdeen's website for more information about avoiding plagiarism before you start the assignment (https://www.abdn.ac.uk/students/academic-life/academic-integrity.php). ***Note that submitting work which is too similar to the work of another student can be considered collusion.*** If you get stuck, please contact your Course Coordinator and ask for help; ***<u>do not ask another student, friend or any other person to do the assignment for you!</u>*** Also, read the following information provided by the University of Aberdeen:
https://www.abdn.ac.uk/sls/online-resources/avoiding-plagiarism/

The University of Aberdeen has a new policy on extension requests, which can be found here:

Policy Extensions and Penalties for Unauthorised Late Submissions of Coursework.pdf

If you want to apply for an extension, please email your request (using your UoA email address) to **uoa-ji-enquiries@abdn.ac.uk** Students should also include any supporting evidence where possible (for example, a medical letter).

Introduction

The aim of this assignment is to demonstrate your skills in Java object-oriented programming. A folder dedicated to the assignment is available on MyAberdeen, containing a code template (*CryptoCoursework*). You are required to complete the existing code according to the specifications provided. The given structure must be preserved: ***you cannot delete any existing code, create new Java files, or add other files to the folder***. Your work should focus on editing and completing the supplied classes. Generally, if you see three dots on a line in the code template (...), it means you need to add one or more instructions in that specific place to complete the class.

Additionally, the assignment introduces the fundamentals of *cryptography*, the science of securing information. You will explore cryptographic algorithms, practice encrypting and decrypting messages, and gain experience in String manipulation. The task also develops your understanding of arrays and their use in storing and processing sequences of elements.

Cryptography

Let us suppose that you want to buy something online. You can use your computer to connect to an online store and choose what you want to buy. Then, you can put your credit card information into your computer, and your computer will send your credit card information across the Internet to the online store. But what if a thief is looking at the data going across the Internet? The thief might be able to intercept your credit card information and use it to make fraudulent purchases. You do not want that, and neither does the online store. So, how can we make online shopping safe?

When buying something online, your computer encrypts the information before it sends it to the online store. Your computer and the online store server need to agree on a special piece of data called the *key*. Then, we can use the key with an encryption algorithm to “transform” the data, so that only another computer with the same key can decrypt it. Thieves can only see the encrypted data but cannot understand what it says.

Historians believe that the Romans created one the first algorithms to encrypt messages and called it *Caesar Cipher*. You will learn about it in this assignment. The basic idea is to substitute each letter with the letter obtained by “shifting” the alphabet by a fixed amount (that is, moving ahead in the alphabet a specific number of letters). The amount you shift the alphabet by is the *key* for this cipher. For instance, Julius Caesar used a shift of 23 letters. To see how this works, consider the following example. We have started by shifting the alphabet below to show how the letters are encrypted.

Alphabet	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Shifted Alphabet	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W

Message: FIRST LEGION ATTACK EAST!

Encrypted message: CFOPQ IBDFLK XQQXZH BXPQ!

The first letter of the message is F. We find the letter F in the alphabet and then we go forward 23 letters (when you get to Z you carry on by start again from A). So, you would write down a C as the first letter of your message. The next letter is I. We find the letter I in the alphabet and then we go forward 23 letters until we reach F, writing down F as the next letter of the message. You would continue the same way through the first word and then, you would get to a space. Doing this by hand as Caesar would have done it, the easiest thing to do is leave the space unchanged and write down a space in your message. For the next word after the space, we can proceed in the same way and so on and so forth until you end up with something that is unintelligible under casual scrutiny. However, if you know, or can figure out the key, you can decrypt the message.

Note that we have pre-shifted the entire alphabet. In other words, we have computed the shifts for each letter at the start, before encrypting anything. Also, note that the process is the same as shifting the alphabet by 3 to the left (that is, encrypting with a key of 26 minus N, where N is the key).

Guidance and requirements

This assessment requires you to complete some Java coding tasks and a written report outlining the steps taken to produce your code. As explained above, you will be provided with a template for the classes you need to help you get started. Your code and the report must conform to the structure described below and include the content as outlined in each section. Each subtask has its own marks allocated. The report ***must provide a full critical and reflective account of the processes undertaken.***

Task 1: Implement a constructor for CaesarCipher (5 marks)

Open the template for the *CaesarCipher* class and look for the constructor. Your first task requires completing the code for the constructor. Initially, this means assigning the value of the key for this cipher to the instance variable *theKey*. You will see comments indicating where to do so—you will also see three dots (...) marking the space.

Whenever we create a new *CaesarCipher* object, we will also generate two String objects: one containing the standard alphabet and another containing the shifted alphabet. This ensures that we can quickly map each letter in the original message to its encrypted counterpart (see the example at the end of the previous page). The shifted alphabet depends on the key, which specifies how many positions each letter should be moved forward to maintain the correspondence.

You will notice that the value for the *alphabet* has already been allocated, but the *shiftedAlphabet* is missing. You must add it at the appropriate place. Recall that the value for the shifted alphabet depends on the key.

Task 2: Implement the Caesar Cipher algorithm (5 marks)

Next, you will complete the method *encrypt*. This method has one parameter, a String named *message*, and returns a String that corresponds to the encrypted message using the Caesar Cipher algorithm. The encrypted message must be kept in a StringBuilder called *encrypted*. At this point, you can assume that all the alphabetic characters in the message are uppercase letters. so, the call encrypt("FIRST LEGION ATTACK EAST!");

should return the string

"CFOPQ IBDFLK XQQXZH BXPQ!"

The template will remind you to check if the characters in the message are alphabetic or not. Non-alphabetic characters (spaces, punctuation, and numbers) must remain unchanged in the encrypted message.

Task 3: Implement testCaesar (5 marks)

You will find a class called *Test* in the *CryptoCoursework* folder. Your task is to complete the methods *testCaesarString* and *testCaesarFile*, both of which are declared as void and take one parameter.

- *testCaesarString* receives a message as a String parameter, creates a new CaesarCipher object with key = 23, encrypt it using the Caesar Cipher, and print out the result.
- *testCaesarFile* receives as a parameter the name of a text file. Then, it creates a new CaesarCipher object with key = 5, reads the contents of the file, encrypts the whole text using the Caesar Cipher, and prints out the encrypted message on the screen.

The template will tell you in which order you should implement these methods. Simply, follow the instructions and add your code in the lines where you see three dots (...). For example, in the case of *testCaesarFile*:

- Create a new *CaesarCipher* object with key 5.
- Read a text file a keep its contents in a String called *message*.
- Encrypt the *message*. Recall which parameters you must pass to the method *encrypt*.
- Finally, print out the encrypted message on the screen.

If you wanted to run the code that you have produced up to this point, you should execute the Main class. However, **you must not modify the code of the Main class.** The code already contains the necessary calls to execute the methods of the *Test* class and print the encrypted Strings on the screen. Also, make sure that the file *"titus_small_uppercase.txt"* remains in its original location within the Cryptography folder on MyAberdeen, so that your program can retrieve it correctly when executed. All required changes should be made within the methods of the *Test* class, *without changing the given template*.

Task 4: Uppercase and lowercase (5 marks)

We will now make some enhancements to ensure that the encryption works with all letters (both uppercase and lowercase). We will do so by modifying the *encrypt* method implemented in Task 1. For example,

encrypt("First Legion");

should return "Cfopq Ibdflk" if the key used is 23;

and

encrypt("First Legion");

should return “Wzijk Cvxzfe” if the key used is 17.

Think carefully how you will ensure that the encryption works with all letters. You must be able to do it by slightly amending your own code for the constructor of the CaesarCipher class.

> *Observation: There are many ways to implement this task. However, all that you need is to make sure that for every letter in the alphabet (lowercase or uppercase), you can “quickly” identify its right replacement for the encrypted message. So, one possible option is to have all the letters in lowercase and uppercase together and then shift them all using the same key. You can try this option at the point where the comments refer to Task 4.*

Task 5: Two keys simultaneously (10 marks)

Historically, once the Caesar Cipher became widely known, people developed new variations to make messages harder to decrypt. One of such variations involved using more than one key simultaneously. In this task, we will implement this alternative using two different keys.

Open the *CaesarCipherMultipleKeys* class and amend the constructor so that it receives an array of keys rather than a single key. For this task, you can assume that the array contains exactly two keys, but you must assign these keys to the instance variable *keys*, and you must use the instance variable within the class. Ensure that the method works with all letters (both uppercase and lowercase).

> *Observation: Given that we have multiple keys, we can agree that we also have multiple shifted alphabets. In fact, we should have one shifted alphabet for each key. Thus, shiftedAlphabet should no longer be a String, but an array of shiftedAlphabet’s with as many entries as keys.*

Next, complete the method *encryptWithMultipleKeys*, which receives a String named *message* and uses the integer keys in the *keys* array to encrypt it according to the following algorithm:

- Use the first key to encrypt every other character in *message*, starting with the first character.
- Use the second key to encrypt every other character, starting with the second character.

For example, the following code

```
int[] keys = new int[]{23, 17};
encryptWithTwoKeys("First Legion", [23, 17]);
```

should return “Czojq Ivdzle”. Note that the F is encrypted with key 23, the first i with key 17, the r with key 23, the s with key 17, and so on.

Task 6: Implement another test (5 marks)

Returned to the *Test* class in the Cryptography folder and work on the method called test *testCaesarMultipleKeysString*. This method should receive a String parameter (the message), create a new *CaesarCipherMultipleKeys* object with two keys (23 and 17), and encrypt the message. At the end, the method should print out the result on the screen.

The best way to approach this task is to look at the comments in the template for the *Test* class and fill in the lines containing three dots *(...).* Be careful when specifying the keys (23 and 17).

Cracking the Caesar Cipher

Languages exhibit predictable patterns. In English, for example, letters such as ‘E’, ‘T’, and ‘A’ occur frequently, while letters like ‘Z’ are rare. The letter ‘E’ alone accounts for about 12% of all characters in English text, whereas ‘Z’ appears less than 0.1%. If every ‘E’ is replaced with ‘X’, then ‘X’ will appear in the encrypted text about 12% of the time. In other words, the statistical distribution of letters does not change (it is merely disguised). Attackers exploit this property by comparing the frequency of symbols in the encrypted text with known frequency tables for the English language. For instance, if ‘Q’ is the most common symbol in the encrypted text, it is a strong candidate for being the letter replacing ‘E’. We will use this to automate the cracking, or breaking, of the Caesar Cipher.

Task 7: Determine the frequencies of the characters (10 marks)

Consider now the class *CaesarBreaker*. Here, you will write the code to find the character that occurs most frequently in a message that we want to decrypt. We will assume this is the letter representing ‘e’, because ‘e’ occurs more frequently than any other letter in English text (in Russian, for example, the letter ‘o’ occurs more frequently than ‘e’).

To count the frequency of the letters in the encrypted message, we will use the method *countLettersFrequency*. This method uses an array of 26 frequency counters: one for each letter from ‘a’ to ‘z’. To make everything simpler, you will convert all the characters in the encrypted message to lower case before counting the frequencies.

At the start of the method *countLettersFrequency*, all the counters should be set to zero. As we scan through the text character by character, we check whether the character is a letter. If it is, we determine its position in the alphabet (0 for ‘a’, 1 for ‘b’, …, 25 for ‘z’) and increment the corresponding counter in the array. For example, if we encounter an ‘h’, we increase the counter at index 7; if we encounter an ‘i’, we increase the counter at index 8. Characters such as spaces, digits, or punctuation should be ignored when counting frequencies, because they are not in the alphabet.

Do not proceed beyond this point, unless you understand how *countLettersFrequency* works. We have deliberately removed the comments from the code so that you can work your way through it until you understand how it works.

Task 8: Identify the character with the highest frequency (5 marks)

Your next task is to implement the method *maxIndex()*, which receives an array containing the frequency for each letter of the alphabet and returns the position of the largest value in such an array. The position returned corresponds to the position of the letter that appears most often in the encrypted text. Since ‘e’ is the most common letter in English, we will assume that this letter represents where ‘e’ has been shifted.

Task 9: Find the key (5 marks)

In our indexing system, a = 0, b = 1, c = 2, d = 3, and e = 4. Thus, to find the key, we must calculate the distance between the letter with the highest frequency and 4. In other words:

key = maxIndex(frequencies) – 4

However, if *maxIndex(frequencies)* is less than 4, we should wrap around by adding 26 (that is, the number of letters in the alphabet) to ensure the key is positive.

Use this information to complete the implementation for the method *getKey(encrypted)*. Note that the method is almost completed. Simply follow the comments and fill in the spaces where you can see three dots (...).

Task 10: Decrypt (5 marks)

Decryption must always be possible, since the intended recipient needs to recover the original message. Consider that a shift of 26 is equivalent to a shift of 0. This means that encrypting with a shift of 7 and then decrypting with a shift of 19 must bring us back to the original text, because together they add up to 26 (in other words, there is no shift). This observation is key to understanding how decryption works, and it also gives us a strategy for cracking a cipher. Indeed, complete the implementation for the *decrypt()* method, which receives an encrypted message and returns a String representing the decrypted message.

Task 11: Testing (5 marks)

Return to the Test class and complete the implementation for the *testCaesarDecryptFile()* method. This method should read a file encrypted with a single key and decrypt its contents, using the *CaesarBreaker decrypt()* method which you implemented in the previous task. After that, the method should print the encrypted message on the screen. Once again, follow the guidance offered in the comments, as they will tell you what to do at each state.

Task 12: Decryption with two keys(10 points)

Now that you can decrypt a message that was encrypted with one key, you will add code to be able to decrypt a message that was encrypted with two keys, using ideas from the single key decryption method and the encryption with two keys method that you wrote above (Task 5).

Recall that when using two keys for encryption, *key1* and *key2*, *key1* is used to encrypt every other character, starting with the first of the String, and *key2* is used to encrypt every other character, starting with the second. To decrypt the encrypted String, it may be easier to split the String into two Strings, one String of all the letters encrypted with *key1* and one String of all the letters encrypted with *key2*. Then use the algorithm to determine the key for each String and then use those keys and the two key encryption method to decrypt the original encrypted message.

For example, if the encrypted message was *"Qbkm Zgis"*, then you would split this String into two Strings: *"Qk gs"*, representing the characters in the odd number positions and "*bmZi*" representing the characters in the even number positions. Then you would get the key for each half String and use the two key encryption method to find the message. Note this example is so small it likely will not find the keys, but it illustrates how to take the Strings apart.

Task 13: Report (25 points)

Your report should describe the overall design of your program, as well as the challenges faced during the development. Please describe and justify each step that is needed to reproduce and run your work by using clear descriptive writing, supporting code-snippets and screenshots. When using screenshots please make sure they are readable. Use code snippets and screenshots sparingly; the report is *not* supposed to be a collection of screenshots!

Be concise. Do not be redundant and do not provide irrelevant information in your report. Your report must have a maximum of 1,000 words.

A good report consists of the following sections:

- Introduction
  - This is an overview of the Java program that you develop as part of this assignment. Typically, students who cannot explain what their code is about are students who fail the assignment.
- Implementation
  - A UML diagram describing the relationships among the classes.
- Reflective account
  - A description of the challenges faced while working on this assignment and how you solved them, including what went well and what did not go well. Explain what you have learnt for the future.
- Conclusions
  - A summary of the main points presented in your report.
- References
  - Include references only if your work relies on external sources. For example, if you read a book or an article with further information about image processing. If you have references, your report must indicate what parts are based on which specific sources.

Marking Criteria

- Clarity and readability of the source code, including commenting of the code.
- Technical correctness of the program (that is, the program passes the tests).
- Reproducibility. How easy is it for other programmers to repeat your work based on your report and code?
- Quality of the report, including language, structure, clarity, conciseness, references (if necessary).

Submission Instructions

You should submit a PDF version of your report along with your code. The name of the PDF file should have the form “JC2002_Assignment_<your Surname>_<your first name>_<Your Student ID>”. For instance, “JC2002_Assignment_Smith_John_4568985.pdf”, where 4568985 is your student ID. *You should submit your code on MyAberdeen* (through the submission link).

Any questions pertaining to any aspects of this assessment, please address them to your corresponding Course Coordinators:

Dr Marco Palomino, marco.palomino@abdn.ac.uk
Dr Yining Hua, yining.hua@abdn.ac.uk