# Exercise for the final project for Module 1

Module 1. Java Syntax
Level 28, Lesson 1

AVAILABLE

## Technical task

Create a program that allows you to encrypt and decrypt text using the Caesar cipher. The program must support several operating modes, handle large files, and include input data validation. Optionally, you can add a graphical user interface and statistical analysis to automatically crack the cipher.

### Main goals:

Implementation of the Caesar cipher:

- Creation and use of the alphabet.
- An algorithm for shifting characters according to a given key.

File processing:

- Functionality for working with files (reading, writing).
- Processing large text files.

Input data validation:

- Checking the existence of files.
- Validity of keys.

### Operating modes:

Text encryption:

- An encryption function that takes a file, a key, and writes the ciphertext to a new file.

Text Decryption:

- A decryption function using a known key.

Decryption using the `brute force` method (optional):

- Implementation of the `brute force` method to try all keys until successful decryption.

Decoding by the statistical analysis method (optional):

- Development of a statistical analysis algorithm for automatic keyless decryption using language features.

User interface development:

- Text menu or (optional) graphical interface

**Additional tasks:**

- Handling errors and exceptions.
- Optimized for performance.
- Documentation and testing.

## Cryptology, cryptography and cryptanalysis

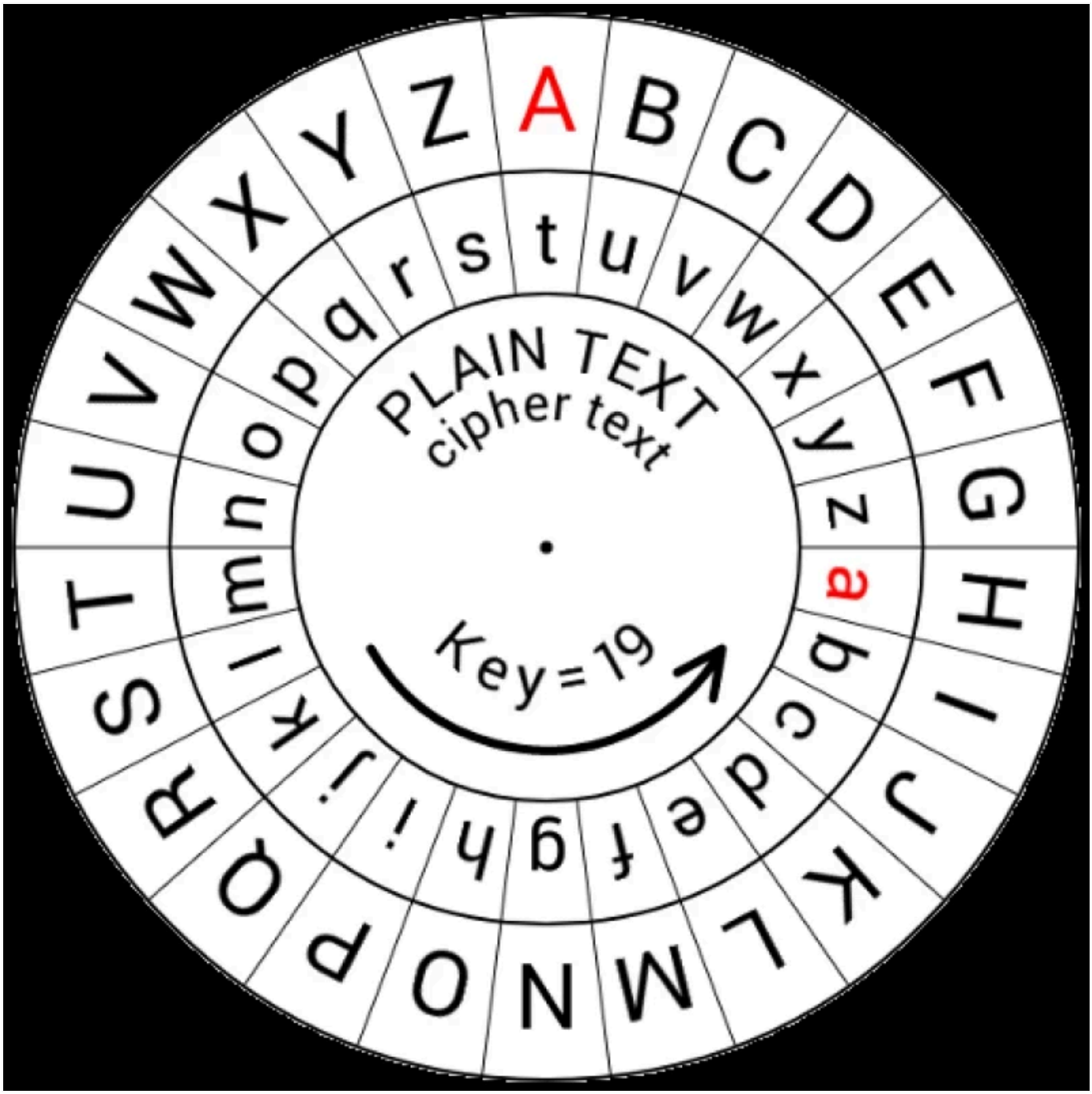
Let's start by analyzing the theory that you will need for the final project. Let’s learn more about cryptology and its components, and at the same time, explore the code that you will use when writing the project.

**Task: Write a program that works with the Caesar cipher.**

The Caesar cipher is one of the simplest and most famous encryption methods. It was named after Emperor Gaius Julius Caesar, who used it for secret correspondence with his generals. The Caesar cipher is a substitution cipher: it replaces each character in the plaintext with a character that is a constant number of positions to the left or right of it in the alphabet.

For example, let's say we set the shift to 3 (key = 3). In this case, A will be replaced by D, B will become E, and so on.

To visualize this, let's look at an encryption tool for the Caesar cipher:




This tool consists of two circles that rotate relative to each other. Along the perimeter of each circle are letters written in alphabetical order. When shifted, all the letters in the inner circle are offset by the same distance relative to the letters in the outer circle.

To encrypt, we replace the letters from the outer circle with the corresponding letters from the inner circle. For example, if the circles are shifted as in the picture, we replace A with C, and R with T. To decrypt, we do the opposite: we replace letters from the inner circle with letters from the outer circle; that is, C back to A, and T back to R.

Let's say Spot the dog decided to write a secret message to his friend Alice. Each of them has such a tool. To successfully encrypt and decrypt the message, they must agree on how many

positions the circles should be shifted relative to each other. In the picture of the tool, you can see that A has become C, and B has become D, which means the inner circle is shifted by 2 positions. Notice that Y became A (and did not disappear). When they agreed to shift the circles by 2 positions and Spot sent Alice the message "NGVU RNCA NGIQ," Alice immediately understood it and went to get a box of Lego.

Since there are only 26 letters in the English alphabet, you can shift the circles by 1, 2,..., 25 positions. If we shift by 26 (or 0) positions, the letters on the circles will coincide.

For more details, you can watch the lecture on CS50 Cryptography, which covers the Caesar cipher and the Vigenere cipher.

**Cryptanalysis:** breaking the cipher

**Cryptography** — data encryption algorithms (and more)

An **alphabet** is a complete set of symbols that can appear in a text that can be encrypted. In the Caesar cipher, the order of these symbols is important. It may not be the classic order of letters in the alphabet, but both Spot (the one who encrypts) and Alice (the one who decrypts) should know it.

Cryptanalysis of statistical data is based on the fact that each language has its own statistics on the use of symbols. For example, in this text (while I am writing it), there are already 14 occurrences of the word "and" and 24 occurrences of the word "in." If you keep the spaces, it’s easy to guess that the most common single-letter words are likely to be A or I. If you used the Caesar cipher with the classic order of letters in the alphabet, deciphering this text would be straightforward (try a shift for A - unreadable - then try a shift for B, and so on).

Okay, you could remove spaces to hide single-letter words. But then you can easily identify vowels (they are much more common than consonants) or frequently occurring words like "or." Even a simple search would not be difficult since there are very few options to try (for the English alphabet, there are only 25). Therefore, Spot and Alice should consider more complex encryption methods.

It is possible to deviate from the literal task. The main thing is to grasp the essence.

This is the minimum theoretical data that you will need to complete the final project. Let's move on to the task description!

## Where to start: program architecture

Before you start writing code, it's essential to break the task into subtasks and think about the overall architecture of the project. Since this is your first big project, you might be tempted to put everything into one big class and create several methods within it, calling them all in the `main` method.

Let's call our class `CaesarCipher`.

In this class, we will define thz alphabet we will work with, as well as methods for encryption, decryption, and a `main` method to handle the call menu.

```
1  public class CaesarCipher {
2
3      // Alphabet
4      private static final String ALPHABET = "our alphabet will be here";
5
6      // Methods for encryption, decryption, brute force, statistical ana
7
8      public void encrypt(String inputFile, String outputFile, int key) {
9      // Implement encryption
10     }
11
12     public void decrypt(String inputFile, String outputFile, int key) {
13     // Implement decryption
14     }
15
16     public void bruteForce(String inputFile, String outputFile,
17 String optionalSampleFile) {
18     // Implementation of brute force
19     }
20
21     public void statisticalAnalysis(String inputFile, String outputFile
22 String optionalSampleFile) {
23     // Implement statistical analysis
24     }
25
26     // Helper methods: validateInput(), createAlphabet(), shiftCharacte
27
28 public static void main(String[] args) {
29     CaesarCipher cipher = new CaesarCipher();
30     // Menu logic
31     // 1. Encryption
32     // 2. Decryption with key
33     // 3. Brute force
```

```
    // 4. Statistical analysis
    // 0. Exit

    // Example of calling the encryption method:
    // cipher.encrypt("input.txt", "output.txt", 3);
    }
}
}
```

But imagine if your program grows. It won't be very convenient to write everything in one class. To better organize the code and increase readability, consider splitting the program into several classes.

For example:

1. `MainApp:` This is the main class where program execution begins. It handles processing user commands, calling appropriate methods, and controlling the program flow.

This approach will help you manage the program more effectively as it becomes more complex.

```
public class MainApp {
    public static void main(String[] args) {
        // Logic for selecting the operating mode, calling the appropri
    }
}
```

2. `Cipher:` Class that implements Caesar cipher and decryption functionality

```
public class Cipher {
    private char[] alphabet;
    public Cipher(char[] alphabet) {
    this.alphabet = alphabet;
    }
    public String encrypt(String text, int shift) {
        // Encryption logic
    }
    public String decrypt(String encryptedText, int shift) {
        // Decryption logic
    }
}
```

3. `FileManager:` Responsible for reading and writing files.

```
public class FileManager {
    public String readFile(String filePath) {
        // File reading logic
    }
    public void writeFile(String content, String filePath) {
        // Logic for writing a file
    }
}
```

4. `Validator:` Validation of input data such as file existence, key validity.

```
public class Validator {
    public boolean isValidKey(int key, char[] alphabet) {
        // Key check
    }
    public boolean isFileExists(String filePath) {
        // Check if the file exists
    }
}
```

5. `BruteForce:` Implementation of a method of enumerating all keys for cracking.

```
public class BruteForce {
    public String decryptByBruteForce(String encryptedText, char[] alph
        // Brute force logic
        }
}
```

6. `StatisticalAnalyzer` For statistical analysis during transcription.

```
public class StatisticalAnalyzer {
    public int findMostLikelyShift(String encryptedText, char[] alphabe
String representativeText) {
    // Statistical analysis logic to determine the shift
    }
}
```

## General approach to using classes:

In MainApp, the user selects the operating mode, for example, through the command line or a simple GUI.

Then, using instances of the `FileManager`, `Cipher`, `Validator`, `BruteForce`, and `StatisticalAnalyzer` classes, appropriate operations such as file reading, encryption/decryption, validation, and analysis are performed. The results are written back to a file or displayed to the user.

These are just recommendations! You can break your program into classes differently, come up with different names, alphabet, and so on.

## Result

### Result: the program works in several modes

Modes:

- Text encryption
- Decrypting text using a key
- Decrypting text using `brute force` (searching through all options)
- (optional) Decrypting using statistical text analysis

The program must open a text file specified by the user and perform one of the above actions with it. After this, it will create a new file with the result.

### The program must perform the following functions:

1. Encrypt Text from a Given File:

- Input: The address of the file with the original text, the address of the file where the ciphertext needs to be written, and an alphabetical shift (the Caesar cipher key).
- Ensure:
  - a) The original file exists at the given address.
  - b) The key is between 0 and (alphabet size - 1) (or you can take the remainder of the division by the alphabet size).

2. Decryption with a Known Key:

- Input: The address of the encrypted file, the address where the decrypted file should be written, and the alphabetical shift used during encryption (key).

3. Decryption Using the `Brute Force` Method:

- Input: The address of the encrypted file, (optional) the address of a file with text that is an example of text that was encrypted (for instance, another work by the same author), and the address of the file that should contain the decrypted text.

4. Decoding by Statistical Analysis:

- Input: The same as for brute-force decryption.

Don't forget to validate the input data. The source text for encryption must be in a file, preferably in .txt format. The program must be capable of handling large texts of hundreds of pages. It should be able to encrypt this file and write the ciphertext to another file.

**Hints**

## Alphabet

Create an alphabet in which the problem exists. By convention, this is the English alphabet and punctuation `. , ” ’ : - ! ?` `SPACE` Don't forget about the space! How can I do that? For example, you can store the alphabet in a `String`, several strings, or an array of `Strings`.

Or... you can create an alphabet based on a `Set`, an array, or a `List`. You can also use an ASCII table.

Remember that the alphabet does not change, so it is logical to make such a variable a `public static final` constant. The name of such variables is usually written in capital letters.

```
1 private static final ListALPHABET = Arrays.asList( 'a', 'b',
2 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
3 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '.', ',', '!', '?',
4 ' );
```

Better yet, use a regular array (since the alphabet does not change, there is no point in putting it in a list). Arrays take up less memory, and therefore, whenever possible, arrays should be used, especially when we are talking about primitive types (in lists, primitive types are wrapped in objects and take up significantly more memory).

```
1 private static final char[] ALPHABET = 'a', 'b', 'c', 'd', 'e', 'f',
2 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
3 'u', 'v', 'w', 'x', 'y', 'z', '.', ',', '!', '?', ' ' );
```

## Encryption

For it you need to know the shift (key) and the alphabet.

For each character of the original text you need:

- check that it is in your alphabet. If it is not there, skip this symbol.
- find its position in the alphabet. Think about what data structure you need to use to speed up this process (every 15 times), because it is not necessary to scan the entire library in search of a book starting with the letter Y (Okay, P).
- find a character at a position shifted by a given offset. And remember that in the example with the toy, Y became A (and did not fly into space). How can this be guaranteed? (you can do (letter position + shift) %( alphabet size). Percentage is the operator for obtaining the remainder of division).
- replace the original character with the encrypted one.

Save the result to a file (to avoid a bad user who will try to mess with your .bash_profile or hosts, validate the output file name!)

## Create a program interface / user menu

The GUI can be created using JavaFX or Swing. It is advisable to bother with this after creating the main program. However, if you don't have time or don't want to spend time on this, you can create a simple text menu and display it in the console. For example:

```
1  import picocli.CommandLine;
2  import picocli.CommandLine.Command;
3  import picocli.CommandLine.Model.CommandSpec;
4  import picocli.CommandLine.Parameters;
5  import picocli.CommandLine.Option;
6  import picocli.CommandLine.ParameterException;
7  import picocli.CommandLine.Spec;
8  import java.util.Locale;
9  import java.io.File;
10
11 @Command(name = "cypher", subcommands =
12 {CommandLine.HelpCommand.class },
13         description = "Caesar cypher command")
14 public class Cypher implements Runnable {
15     @Spec CommandSpec spec;
16     @Command(name = "encrypt", description = "Encrypt from
17 file to file using key")
```

```
    void encrypt(
                @Parameters(paramLabel = "", description =
"source file with text to encrypt") File src,
                @Parameters(paramLabel = "", description =
"dest file which should have encrypted text") File dest,
                @Parameters(paramLabel = "", description =
"key for encryption") int key) {
        // TODO
    }

    @Command(name = "brute force", description = "Decrypt
from file to file using brute force") // |3|
    void bruteForce(
                @Parameters(paramLabel = "", description =
"source file with encrypted text") File src,
                @Option(names = {"-r",
"--representative"}, description = "file with unencrypted
representative text") File representativeFile,
                @Parameters(paramLabel = "", description =
"dest file which should have decrypted text") File dest) {
        // TODO
    }

    @Command(name = "statistical decryption", description =
"Decrypt from file to file using statistical analysis") // |3|
    void statisticalDecrypt(
                @Parameters(paramLabel = "", description =
"source file with encrypted text") File src,
                @Option(names = {"-r",
"--representative"}, description = "file with unencrypted
representative text") File representativeFile,
                @Parameters(paramLabel = "", description =
"dest file which should have decrypted text") File dest) {
        // TODO
}

    @Command(name = "decrypt", description = "Decrypt from
file to file using statistical analysis") // |3|
    void decrypt(
                @Parameters(paramLabel = "", description =
"source file with encrypted text") File src,
                @Parameters(paramLabel = "", description =
"dest file which should have decrypted text") File dest,
```

```
                @Parameters(paramLabel = "", description =
"key for encryption") int key) {
        // TODO
    }

    @Override
    public void run() {
        throw new ParameterException(spec.commandLine(),
"Specify a subcommand");
    }

    public static void main(String[] args) {
        int exitCode = new CommandLine(new
Cypher()).execute(args);
        System.exit(exitCode);
    }
}
```

If you don't want to deal with the Picocli library, you can do it as simply as possible: make a menu using loops or a `switch` statement. It is also necessary that the program terminates at the user's request (for example, by entering the word "exit").

## Working with files

To work with files, it is recommended to use the Java NIO library, since:

- This is a more modern and performant API compared to IO.
- Supports asynchronous file processing.
- Ensures correct work with large files.

**Main NIO classes for working with files:**

- `Path:` Represents the path to a file or directory.
- `Files:` Provides static methods for working with files and directories (read, write, copy, delete, etc.).
- `Charset:` Represents the character encoding used when reading/writing text files.

Example of reading a large file using NIO:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
```

```java
import java.nio.file.Paths;
import java.nio.charset.StandardCharsets;

public class FileHandler {

    public static String readFile(String filePath) throws IOException {
    Path path = Paths.get(filePath);
    byte[] bytes = Files.readAllBytes(path);
    return new String(bytes, StandardCharsets.UTF_8);
    }

    // ... other methods for working with files ...
}
```

**Recommendations for working with large files:**

- Read/write in parts. For very large files that do not fit in RAM, use the `Files.lines()` methods to read line by line or `Files.newInputStream()` to read block by block.
- Buffering. Use `BufferedReader` and `BufferedWriter` to buffer I/O operations to improve performance.

## Decoding

For it you need to know the shift (key) and the alphabet.

For each character of the ciphertext you need:

- check that it is in your alphabet. If not, hackers are breaking you. Panic (or return the error).
- find its position in the alphabet.
- find a character at a position shifted by a given shift (but just remember: you are not trying to encrypt the cipher again, so we shift it in the other direction).
- replace the encrypted character with the decrypted one

You will use this code in the following subtasks, so you can output the result to the stream.

You need to save the result to a file.

## Hacking (Brute Force)

You can use the code you wrote to decrypt with a known key, substituting all possible key values.

But how do you know if it was possible to decipher? Use example text (representative text by the author or in the same style). You can compile a dictionary of words and create a metric based on how many words match and how long they are; or another metric that studies the length of words and sentences, or look at which letters most often precede which letters or a dictionary of the most common beginnings of a word (3 letters), you can not use any representative files at all and check the correctness of punctuation and spaces; Save the option with the best results to an output file.

## Hacking (Statistical Analysis)

*Additional requirements(optional)*

Use example text (representative text by the author or in the same style) and compile statistics of letters (for example, how often they appear per 1000 characters). By the way. It is easy to crack the cipher without such a file and analysis: try to guess the space - this is probably the most common character in plain text.

Next, compile the same statistics for the ciphertext. Please note that simply counting characters is not enough since texts can be of different lengths!

Next, calculate the deviation for each possible shift of the encrypted statistics relative to the representative one (for this you can use the sum of squares of the deviation or the dot-product of vectors). Find the shift that gives the minimum deviation and decipher using this shift.

*The project is checked as the group goes through it*

‹ Previous

Quest map ›

Activity

Affiliate Program

**COMPANY**

About us

Contacts

Reviews

Press Room

CodeGym for EDU

FAQ

Support

CodeGym is an online course for learning Java programming from scratch. This course is a perfect way to master Java for beginners. It contains 1200+ tasks with instant verification and an essential scope of Java fundamentals theory. To help you succeed in education, we've implemented a set of motivational features: quizzes, coding projects, content about efficient learning, and a Java developer's career.

**FOLLOW US**

**INTERFACE LANGUAGE**

English