

CS 465 - Programming Assignment 1

Kryzstof Kudlak - 800294138

January 31, 2022

Status

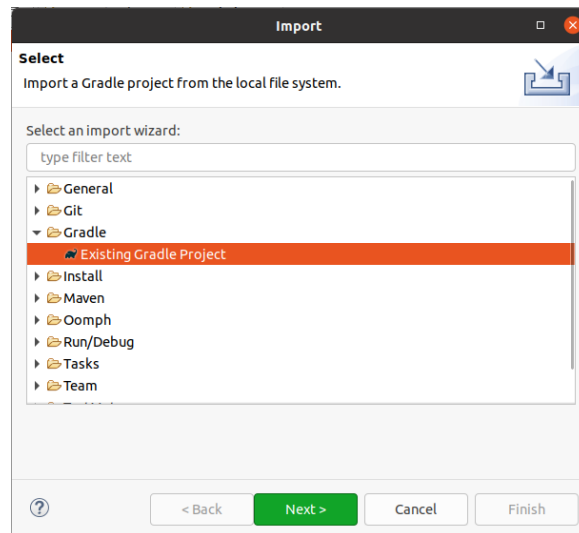
My program compiles correctly and appears to work completely as intended given the 3 test cases and friendly input when selecting a test case to evaluate at the beginning of execution.

Programming Environment

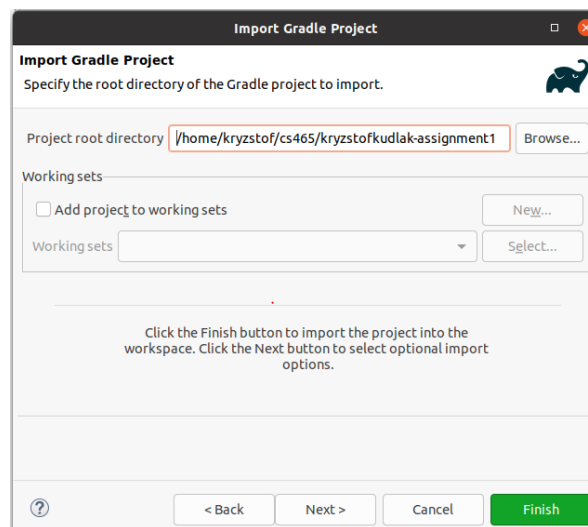
For this assignment, I stuck to what I'm familiar with and wrote it in Java as a Gradle project. I used the LOUD VM version 20.04 (<https://lcseesystems.wvu.edu/services/loud>) and created my project using the default installation of Eclipse along with the accompanying default JDK (16) and Gradle version (7.4).

To import my project into Eclipse, first open Eclipse and select **File > Import...**

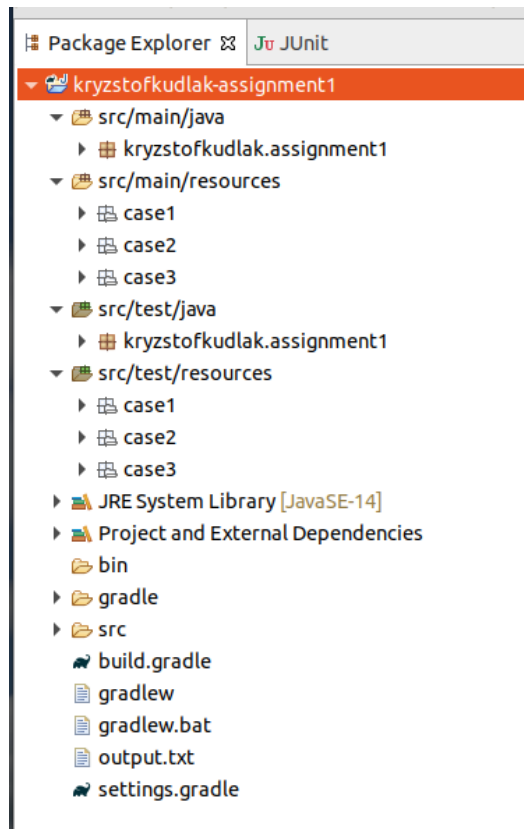
Next, select **Gradle > Existing Gradle Project** from the popout menu. Then click the **Next >** button.



On the next page, select **Browse...** next to the project root directory input space, and find my project on your machine.



Finally, click **Finish**. The project should look like this:



Despite the fact that I completed this assignment as a Gradle project, I did not use any external dependencies besides JUnit 4.12. Everything in `src/main/java` should be able to compile and run using only the default dependencies provided by JDK 16.

The overall structure of my project is pretty standard:

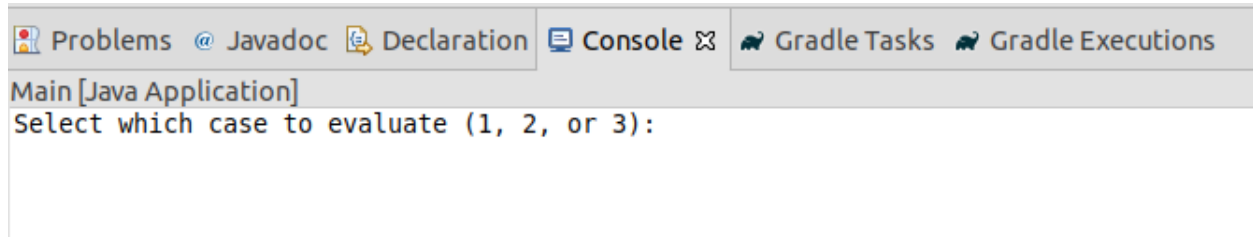
- `src/main/java`: Everything important that is necessary to execute the program
- `src/test/java`: Unit tests that I used as I went along to verify that everything was working as intended
- `src/main/resources`: `input.txt` and `key.txt` files for each of the test cases we were given. The test cases are separated by folder.
- `src/test/resources`: `expected_output.txt` files for each test case which match the `output.txt` files we were given. I originally intended to work these into the test classes, but I didn't get around to it.
- `output.txt`: The output file generated after executing my program
- `build.gradle/settings.gradle`: Used to configure Gradle

The main method of my program exists in `Main.java` which can be found in the `src/main/java` package `kryzstofkudlak-assignment1`. A full run of the program is executed by running this file as a Java application.

Program Description

Main.java

The program starts execution by running Main.java as a Java application. It first prompts the user to specify which test case they would like to run.



If the user enters anything other than 1, 2, or 3, the program will throw an exception and crash. This is intentional as the target user is expected to be competent enough to give appropriate input, and it was not required that I implement this feature in the first place. Please just give it friendly input.

After the user selects which case to test, the application pulls the input.txt and key.txt files from the chosen “case” directory. I effectively just insert whatever the input is into the file path “case_/input.txt” and create an input stream with that file.

Next, I create the file “output.txt” in the root directory of the project as well as an output stream to write to it. Using the OutputStream class, I am able to effectively treat the file like it’s System.out.

```
// setup I/O streams
InputStream inputStream = ClassLoader.getResourceAsStream("case" + selection + "/" + IN_FILE);
String input = new String(inputStream.readAllBytes());
inputStream.close();

InputStream keyStream = ClassLoader.getResourceAsStream("case" + selection + "/" + KEY_FILE);
String key = new String(keyStream.readAllBytes());
keyStream.close();

File outFile = new File(OUT_FILE);
outFile.createNewFile();
OutputStream output = new FileOutputStream(outFile, false);
```

Finally in main(), there’s a giant chunk of method calls to an Encryptor class I made as well as OutputStream.write() statements to format the output file. The ordering of all these statements is pretty much one-to-one with what the assignment doc expects.

Encryptor.java

Encryptor drives the stages of encryption according to what is defined in the project doc. Each stage X listed in the doc has a method within Encryptor called “performX()”. Each of these methods carries out the logic necessary to accomplish that stage of encryption.

performPreprocessing(String input)

This method takes an input string, transforms it into an IntStream of chars, filters out anything that isn't [A-Z], and returns the result back as a string. This accomplishes the preprocessing stage which requires that I strip out any whitespace, punctuation, etc. from the input.

performSubstitution(String key, String input)

For this method, I created two helper classes: Pair and VignereCipher. A “Pair” as I’ve defined it is just a storage class for two variables of any type that I want to couple together. In this method, I begin by iterating over the input string and coupling each character with its respective key character as a “Pair.” Characters in the input string are matched up with characters from the key string using some simple modulo math.

The VignereCipher class has an underlying static array that stores the Vignere Cipher table. I gave this class a method called substitute(key, value) which returns the result of a Vignere Cipher substitution given the key and value which it obtains simply by accessing the underlying array. This method is called for each Pair created in step 1, and the results of each method call are used to build the string to return.

performPaddingAndCreateBlocks(String input)

This method creates a list of Block given an input string. A Block is a class that contains a 4x4 array of characters along with methods to manipulate those characters. For each set of 16 characters in the input string, a Block is created and added to Encryptor’s list of Blocks. The Block constructor works by initializing the underlying array to contain entirely ‘A’s, then uses the input string (expected to be no more than 16 characters) to fill up the array. For this reason, this creation of Blocks fulfills the requirements of the performPadding task.

performShiftRows()

This method iterates over the list of blocks stored in Encryptor and calls the shiftAllRows() on each.

The shiftAllRows() method within Block iterates over each row in block and calls shiftRows(int rowNumber, int byHowMany) to cyclic right shift a row by a specified number of units. It shifts the first row by 4, the second by 3, and so on.

The shiftRows(int rowNumber, int byHowMany) method within Block performs a cyclic right shift on a row given the number of units to shift by. This is done by first copying the row into a new

array, then moving each element of that array to its (current index + byHowMany) % BLOCK_SIZE in the underlying 4x4 array.

performParityBit()

For each Block in Encryptor's list of Blocks, this method calls the setParity() method.

The setParity() method within Block iterates over the entire underlying 4x4 character array, checks if it has an odd number of ones using a helper method, and flips the MSB if so. The MSB is flipped by taking the XOR of the character and the mask 1000 0000.

hasOddNumberOfOnes(int input) is a helper method within Block which accepts a character and returns true if its binary representation has an odd number of ones. The way this is determined is I count the number of ones in the binary representation, then return whether that value % 2 is > 0. The number of ones are counted by right shifting the character and checking its LSB until the character is 0. Each time the LSB is found to be set, I increment the value that stores the total number of 1s found.

performMixColumns()

For each Block in Encryptor's list of Blocks, this method calls the mixColumns() method.

The mixColumns() method in Block iterates over each column in the underlying 4x4 array and calls mixColumn(int colIndex) on it.

The mixColumn(int colIndex) method follows the description in the document verbatim because I didn't really understand why the operations needed to happen. I think it's readable enough to where you can just look at it compared with the document and understand exactly what it does:

```
private void mixColumn(int col) {
    int c0 = this.block[0][col];
    int c1 = this.block[1][col];
    int c2 = this.block[2][col];
    int c3 = this.block[3][col];

    int a0 = RGF.mul(c0, 2) ^ RGF.mul(c1, 3) ^ c2 ^ c3;
    int a1 = c0 ^ RGF.mul(c1, 2) ^ RGF.mul(c2, 3) ^ c3;
    int a2 = c0 ^ c1 ^ RGF.mul(c2, 2) ^ RGF.mul(c3, 3);
    int a3 = RGF.mul(c0, 3) ^ c1 ^ c2 ^ RGF.mul(c3, 2);

    this.block[0][col] = (char) a0;
    this.block[1][col] = (char) a1;
    this.block[2][col] = (char) a2;
    this.block[3][col] = (char) a3;
}
```

As you can see, I created the class RGF and gave it a static mul(int input, int byHowMuch) method so that I could write formulas here that look exactly how they're shown in the project

doc. This method accepts any byte as its first parameter, but it only accepts 2 or 3 as its second parameter. If it doesn't receive 2 or 3, it throws an `IllegalArgumentException`.

The multiplication steps performed by `RGF.mul(int input, int byHowMuch)` are exactly as defined in the project documentation. It begins by left shifting the input by 1 and storing it in a new integer called result. This is the "multiply by 2" step and happens in both cases. Next, if `byHowMuch` is 3, we perform "result XOR input" which adds x (the input) to the result. Then, if the MSB is set, I perform "result XOR 0001 1011" because the assignment doc said to. Finally, because Java uses 4 byte integers, I return "result & 1111 1111" so that only the least significant byte is returned.

Outputting Data

Both the `Encryptor` and `Block` classes have methods to send data to the `OutputStream` that is given to their print methods. In specific:

`Encryptor.printBlocks(OutputStream output)` - For each `Block` in the underlying list of `Blocks`, call that `Block`'s `printBlock` method.

`Block.printBlock(OutputStream output)` - Iterates over each character in the underlying 4x4 array and sends it to the output stream. The thing that's outputted looks like this:

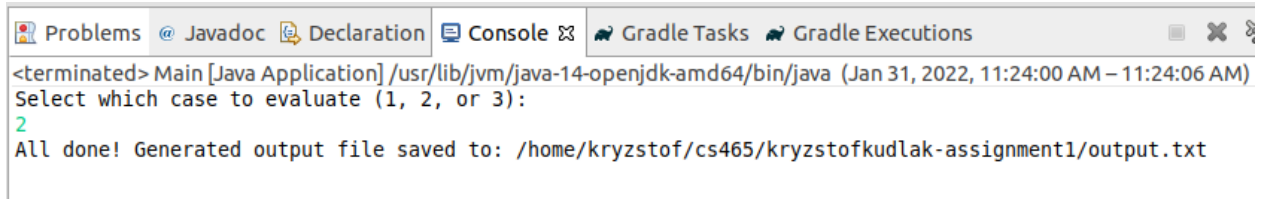
```
ASDF
JKLM
NOPQ
RSTU
```

`Encryptor.printBlocksAsHex(OutputStream output)` - For each `Block` in the list of `Blocks`, call that `Block`'s `printBlockAsHex` method.

`Block.printBlockAsHex(OutputStream output)` - Iterates over each character in the underlying 4x4 array, converts it to pretty hex using `Integer.toHexString()`, and sends results to the output stream. The thing that's outputted looks like this:

```
a0 b9 c4 12
51 6c 4d ef
e3 d4 4d 66
67 69 04 c3
```

By default, all output is sent to `output.txt` which can be found in the root directory of the project. For extra help, the absolute path where the file has been saved is outputted to the console after the program has completed execution:



The screenshot shows the bottom portion of an IDE window with several tabs: Problems, Javadoc, Declaration, Console, Gradle Tasks, and Gradle Executions. The Console tab is active, displaying the output of a Java application. The text in the console indicates the application has terminated, shows the main class and JVM path, prompts for a case evaluation, receives the input '2', and reports the successful generation of an output file.

```
<terminated> Main [Java Application] /usr/lib/jvm/java-14-openjdk-amd64/bin/java (Jan 31, 2022, 11:24:00 AM – 11:24:06 AM)
Select which case to evaluate (1, 2, or 3):
2
All done! Generated output file saved to: /home/kryzstof/cs465/kryzstofkudlak-assignment1/output.txt
```