CI401 - ATM Project

# Project Description

My project was to code a functional ATM program. This program would be able to hold bank accounts, that users can log into. Once logged in, users will have the option to check their balance, withdraw, deposit, and change their password.

I decided to focus on developing modules that will increase the security of the ATM. To do this, I incorporated encryption into the login process. So instead of comparing plain text passwords to login, the program compares their cypher text counter parts. My goal of this was to make the storage of sensitive user data as secure as possible.

# How to run the project

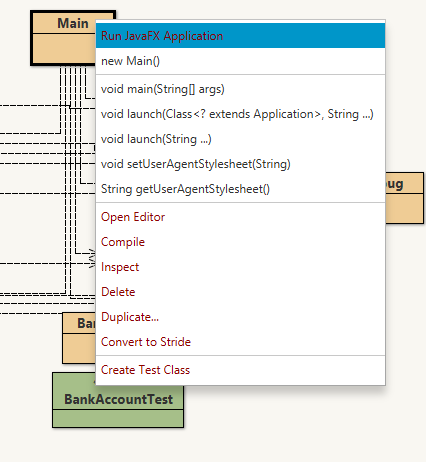
Run the program by right-clicking on the *Main* class and clicking *Run JavaFX Application*.

Figure - Run Project

Default Login Values:

Acc\_no: 10001

Acc\_pw: 11111

# How the code works

The first class to execute code is *Main*.

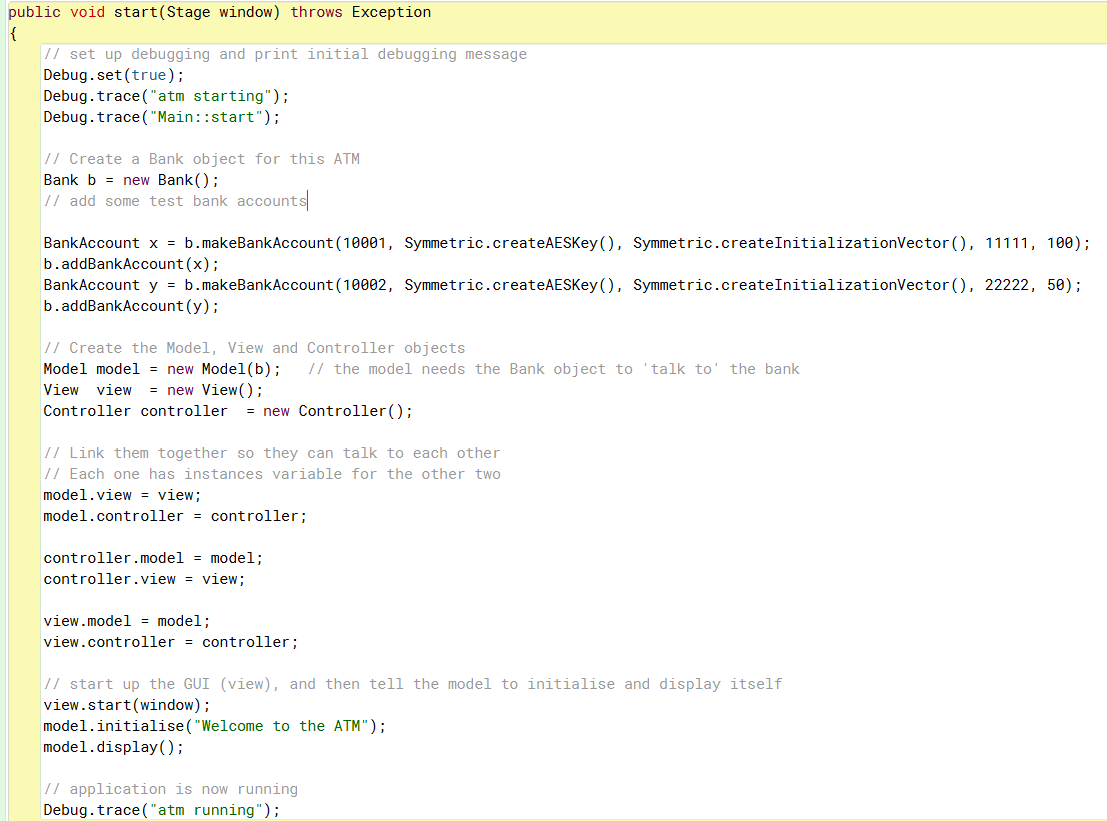
 BlueJ automatically calls the *start* method in the *Main* class. Here, *Main* constructs a new *Bank* object and creates 2 default *BankAccount* objects. It also creates instance variables that link the *Model*, *Controller*, and *View* classes. Then, it starts up the GUI by calling the *view.start()* method.

Figure - Main Class

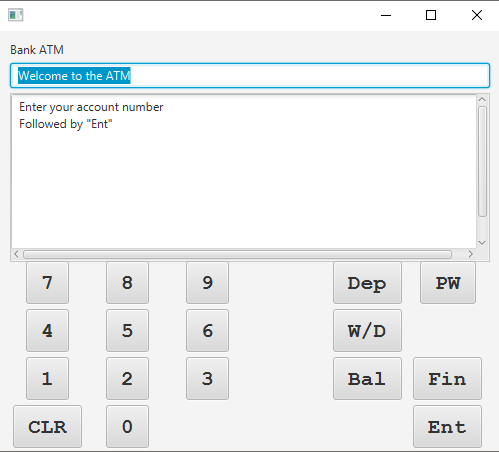


Figure - ATM GUI

After the GUI has loaded, the program waits in it’s first state: ACCOUNT\_NO. In this state, the program waits for the user to press ‘Ent’ after inputting an account number.

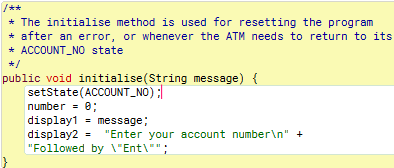


Figure - Initialise Method

After the user hits ‘Ent’, the *Model* class executes it’s most crucial method, *processEnter()*. The input is saved and the state switches.

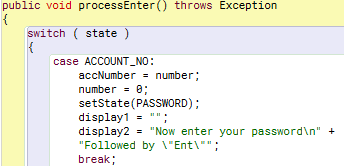


Figure - When enter is pressed in ACCOUNT\_NO state

Now, the program is awaiting the password, followed by ‘Ent’.

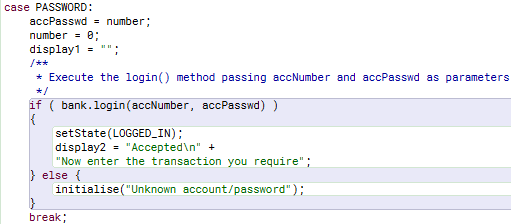


Figure - When enter is pressed in PASSWORD state

After the program detects ‘Ent’ has been pressed, the user input is saved and the *bank.login()* method is executed.

This method iterates through all bank accounts stored in *accounts[]*, and checks if the user inputted account number matches any of the account numbers belonging to the accounts. If one matches, then the method creates new encryption variables.

The method compares the cipher text version of the password stored in the account object; and a cipher created using the accounts secret key and initialisation vector, along with the password the user just entered. If both cipher texts match, then we know the password just entered matches that account.

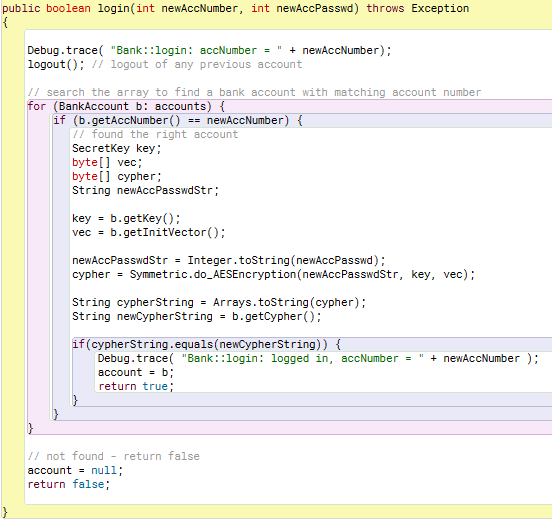


Figure - Bank login method

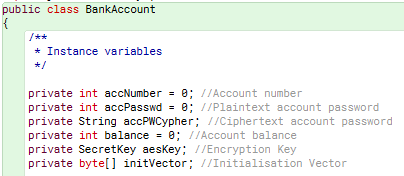


Figure - BankAccount instance variables

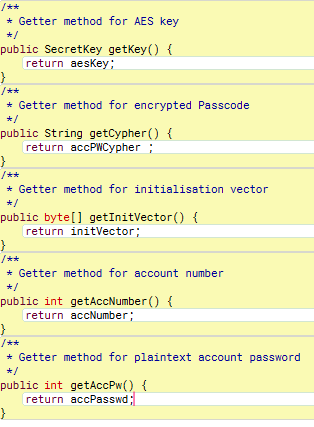


Figure - BankAccount getter methods

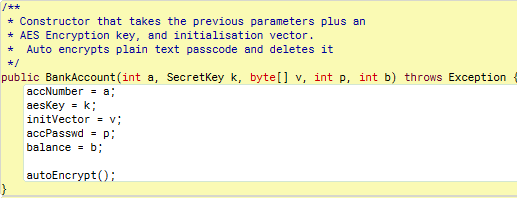
The use of encryption was an attempt to make the program more secure, so that plain text passwords were used as sparingly as possible. Another way I implemented **this philosophy was in the BankAccount constructor.

Figure - BankAccount constructor method

*autoEncrypt()* is a method that automatically converts a plain text password into its cipher text counter-part, before removing the plain text version completely. I hoped to make it more difficult for a hacker to compromise sensitive user data if I store it in cipher text.

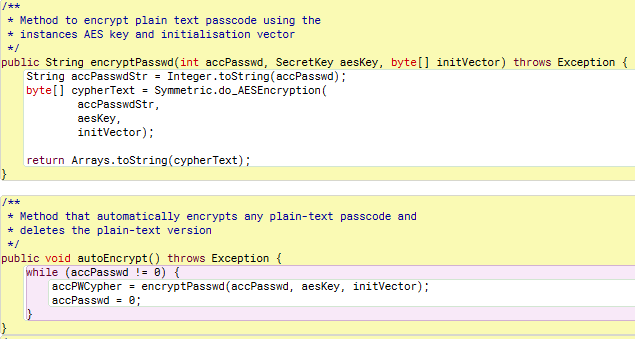


Figure - autoEncrypt() and encryptPasswd() methods

These methods use the *Symmetric* class, this is where all the cryptographic logic for the program is stored.

The other main functionality I added to the ATM was the ability to change the accounts password. Once logged in, the user can press the ‘PW’ button to enter the PASSWD\_CHANGE state. Once in this state, the program waits for the user to ‘Ent’ a new password. Once the *Model* has checked that the password is between 4-8 characters long, the user enters the new password once again, to confirm they had not typed it incorrectly. If successful, the password is changed and the *autoEncrypt()* is automatically called.

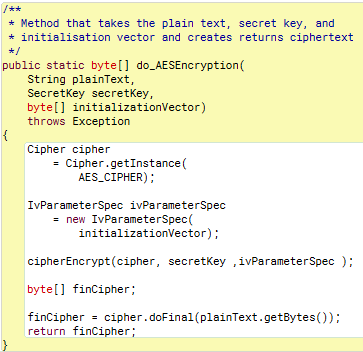


Figure - Symmetric encryption method

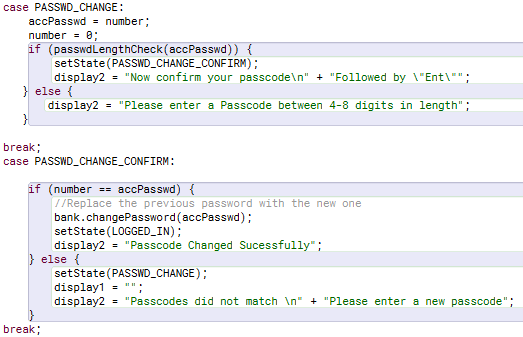


Figure - processEnter() password change

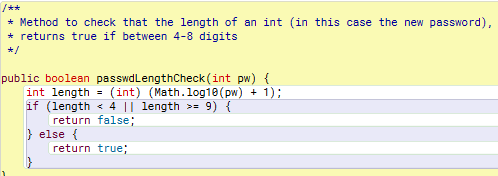


Figure - Password Length Checker

# Testing



Figure - Failed Test

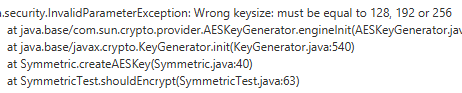


Figure - Test Error Dialogue Box

I have used jUnit testing to help troubleshoot issues in my program, and to create automatic tests that can continue to test that components of the logic are working as intended. By following the stack trace of the error, I found that it was the key size parameter in this method that was causing the error. After changing the number to 256, the test now executes successfully.

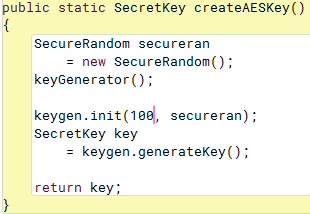


Figure - Culprit of Error



Figure - Test now successful

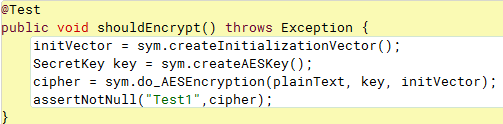


Figure - Test method

I created 12 methods to test the basic ATM functionality. One of these methods, I have been unable to make execute successfully, due to a *NullPointerError*. This is an issue I have had while trying to test elements of the UI.

# Reflection

Overall, I think I did a good job of making the program more secure, and sensitive user data more obfuscated. All the functionality I have added works as intended. So, to that end I think I have achieved my goal for this project. However, there are still elements I would have liked to improve.

I would have liked to fix the *NullPointerError*, so I could write successful tests for the UI elements, as opposed to just the business logic. I also would have like to change the UI to match that of a real ATM better (buttons that do different actions depending on the context).

# References and Research

<https://www.geeksforgeeks.org/symmetric-encryption-cryptography-in-java/>

(Symmetric Encryption)

<http://junit.sourceforge.net/javadoc/org/junit/Assert.html>

(Java assertions)

<https://marcus-biel.com/access-and-non-access/>

(Java OO concepts)