Objectives

In this assignment, you will learn how to use the Java Cryptography Extension (JCE) to:

- perform symmetric encryption and decryption
- demonstrate CPA security
- implement password-based encryption
- designing for changes in recommended key length and algorithms

Part 1: Perform symmetric encryption and decryption (45%)

Extend the existing FileEncryptor.java to allow the user to specify:

- Encryption or decryption operation.
- The secret key in Base64 encoding (for decryption mode).
- Input file and output file paths.

> Note that the code above uses a Util class available from here.

The encryption operation is indicated by the keyword `enc` and the decryption operation is indicated by the keyword `dec`.
A secret key is randomly generated and output using Base64 encoding (Base64 JavaDoc). The secret key can also be provided for decryption encoded using Base64.
The cipher mode will remain the same as in the example code `AES/CBC/PKCS5PADDING`.
Some examples:

```
% java FileEncryptor enc plaintext.txt ciphertext.enc
Secret key is ((base64 encoded key))

IV is ((base 64 IV)
```

This will encrypt the file `plaintext.txt` as `ciphertext.enc` and printout the randomly generated secret key encoded as a base64 string.

```
% java FileEncryptor dec ((base 64 encoded key)) ((base 64 IV))
ciphertext.enc plaintext.txt
```

This will decrypt the file `ciphertext.enc` as `plaintext.txt`.

Part 2: Demonstrate Chosen-plaintext (CPA) Security (30%)

The code above incorporates an IV to provide chosen-plaintext (CPA) security for our files but because the key is randomly chosen it is impossible to demonstrate.

Extend your program as follows:

- Can specify the secret key as a base 64 string
- No longer need to specify the IV when decrypting.

You should also be able to demonstrate that you can successfully decrypt a previously encrypted file and that the ciphertext is different each time. To demonstrate this you could use the utility hexdump with the `-b` flag.
Note that is the same as part 1 but the IV needs to be stored (note that here the IV plays the role of a salt). You need to consider whether to keep the IV secret or not and how to achieve that. Spoiler - you don't have to encrypt the IV but you should explain why in your write up about the design
Some examples:

```
% java FileEncryptor enc ((base 64 encoded key)) plaintext.txt
ciphertext.enc
```

This will encrypt the file `plaintext.txt` as `ciphertext.enc`.

```
% java FileEncryptor dec ((base 64 encoded key)) ciphertext.enc
plaintext.txt
```

This will decrypt the file `ciphertext.enc` as `plaintext.txt`.

Part 3: Generating a Secret Key from a Password (10%)

Keys are hard to remember so using a password is a better option. You could generate a key directly from this but the user might choose a password with low entropy making defeating the encryption scheme trivial. A better approach is password-based encryption where you add salt, iterate, and hash repeatedly. Java provides algorithms to make this easy, see here for a code example. This follows NIST recommendations (found here PBKDF2 and here RFC2898). You should also read the OWASP advice on password storage.
This requires adding an additional option to `FileEncryptor` to allow a password to be specified. We also ask that you continue to output the secret key as generated from the password for marking purposes.
Some examples:

```
% java FileEncryptor enc "my password" plaintext.txt ciphertext.enc
```

This will encrypt the file `plaintext.txt` as `ciphertext.enc` to derive a secret key from "my password" and printout the randomly generated secret key encoded as a base64 string.

```
% java FileEncryptor dec "my password" ciphertext.enc plaintext.txt
```

This will decrypt the file `ciphertext.enc` as `plaintext.txt` using a secret key generated from "my password".

Part 4: Designing for changes in recommended key length and algorithms (15%)

Algorithms and recommended key lengths change over time but you may have legacy files on your system.
Extend your program to:

- Allow the user to specify the algorithm (Blowfish and AES) as well as the key length.
- Embed metadata recording the algorithm and key length used to encrypt the file.
- Use the metadata when decrypting to pick the correct algorithm and key length to use.
- Allow the user to query the metadata embedded in the file.

This requires adding two additional options to `FileEncryptor` to allow the algorithm and key length to be specified when encrypting a file and another option `info` for printing the metadata. Some examples:

```
% java FileEncryptor enc AES 128 "my password" plaintext.txt
ciphertext.enc

Secret key is ((base64 encoded key))
```

This will encrypt the file `plaintext.txt` as `ciphertext.enc` to generate a 128-bit key compatible with AES based on "my password" and printout the randomly generated secret key encoded as a base64 string.

```
% java FileEncryptor dec "my password" ciphertext.enc plaintext.txt
```

This will decrypt the file `ciphertext.enc` as `plaintext.txt` using "my password" derived secret key and the metadata embedded in `ciphertext.enc`.

```
% java FileEncryptor info ciphertext.enc

AES 128
```

This will print the metadata contained within `ciphertext.enc`.

submission:

- Code - submit the most complete and correct version of FileEncryptor with comments to help the marker identify your changes and help them understand your code. The code should output human-readable error messages that will help users correct their mistakes rather than just providing a stack trace.

- A file named README containing the documentation a write-up describing your design choices and documenting testing that shows that your code meets the requirements (doesn't need to be automated). At a minimum, you should document part 2 and part 4 and explain why your design is secure.

The relative proportion of marks per part of the assignment is shown above and within these parts your system will be graded on two bases:

- Does it work? roughly 80 percent of marks
    - We will check to make sure that your system works.
    - Your security will not matter for this part of the grade. However, if your encryption/decryption methods fail and cause bad things to happen that either leak information unnecessarily or lead to insecure encryption you will lose points.
- Is it documented and secure?
    - You should record design decisions for each part and at a minimum, your write-up for parts 2 and 4 should explain how they work and why the design is secure. 15 percent of marks
    - We will go through your code and check for security issues. Thus, it would be extremely helpful (and possibly beneficial to your grade) if you clearly comment on your steps to ensure security - this includes thinking about sensible defaults. This blog provides (at the end) some advanced tips with respect to secure Java. 5 percent of marks

Excellent implementation and write up for parts 1-3 will mean a maximum of a low A grade.