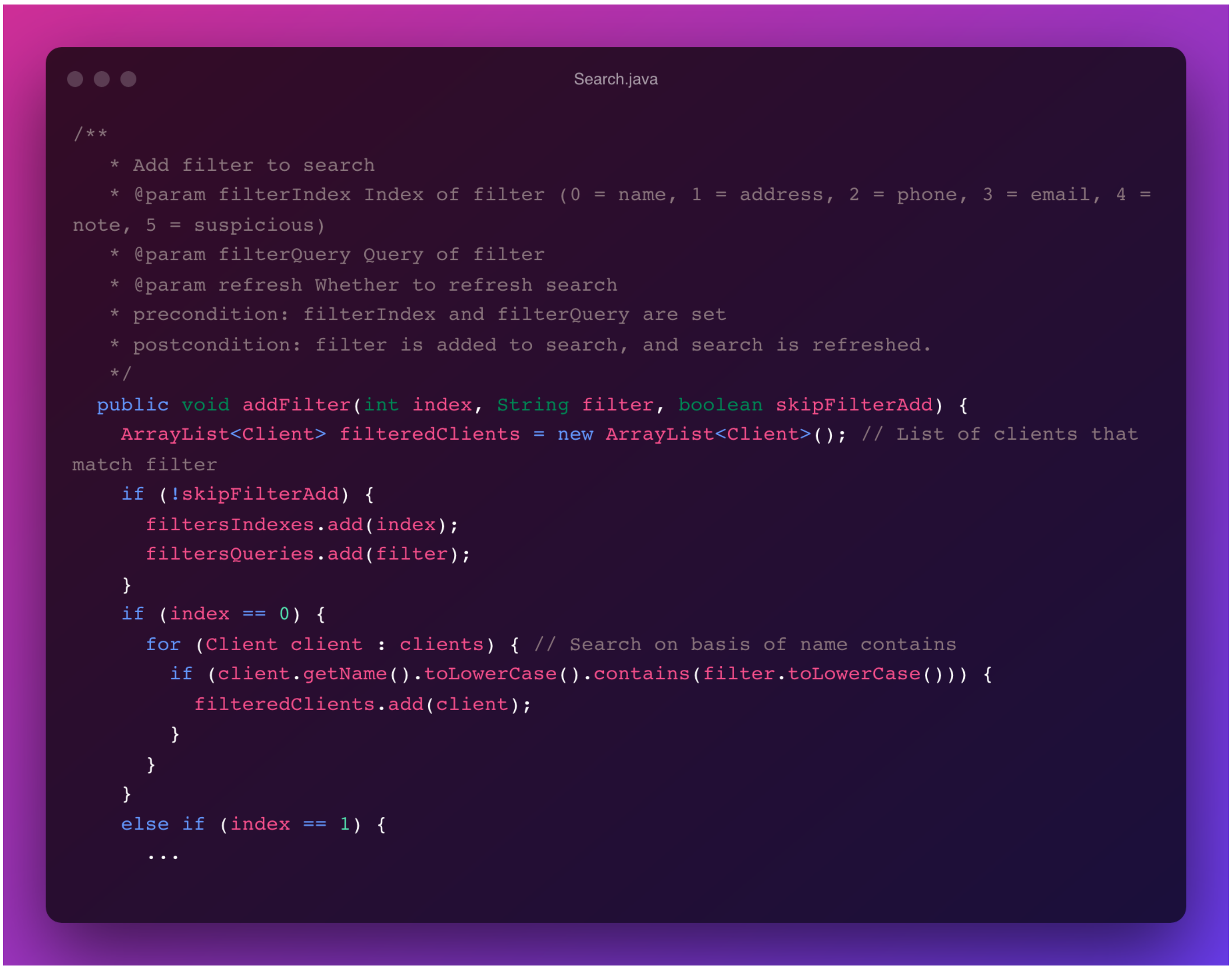
Write-up

Rohin Arya, Y11 HL DP Computer Science, May 31

- Client Search
- Persistent Data Management
- Caesar Cipher Encryption

Client Search

The purpose of including a client search is to meet a requirement set by the client. The search feature allows the user to create a series of "filters" which can blend together to produce a list of clients. The process of adding a filter is shown below:

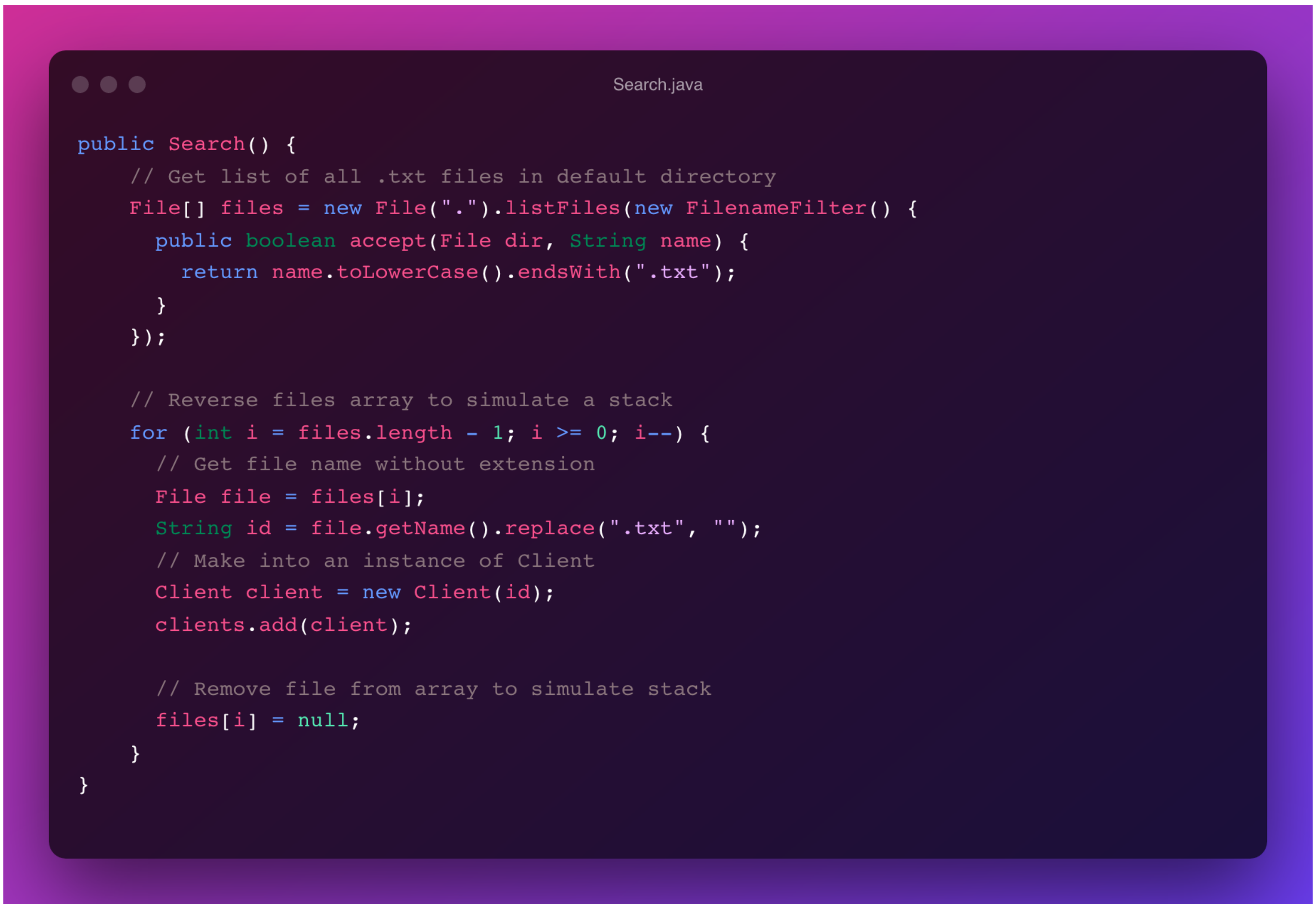


```java
/**
   * Add filter to search
   * @param filterIndex Index of filter (0 = name, 1 = address, 2 = phone, 3 = email, 4 =
note, 5 = suspicious)
   * @param filterQuery Query of filter
   * @param refresh Whether to refresh search
   * precondition: filterIndex and filterQuery are set
   * postcondition: filter is added to search, and search is refreshed.
   */
  public void addFilter(int index, String filter, boolean skipFilterAdd) {
    ArrayList<Client> filteredClients = new ArrayList<Client>(); // List of clients that
match filter
    if (!skipFilterAdd) {
      filtersIndexes.add(index);
      filtersQueries.add(filter);
    }
    if (index == 0) {
      for (Client client : clients) { // Search on basis of name contains
        if (client.getName().toLowerCase().contains(filter.toLowerCase())) {
          filteredClients.add(client);
        }
      }
    }
    else if (index == 1) {
      ...
```

The `addFilter` method includes three parameters: index, filter, and skipFilterAdd). The index parameter determined which field to match the filter parameter to. The final boolean parameter determines whether to skip adding the filter to the list of filters. When refreshing the search (reading from the list of filters and applying them), this is set to `true` in order to prevent double counting each filter.

Next, for every client in "clients" (`Client client : clients`), the relevant client data is fetched using a "get" method in the Client class and if it includes the search query (`filter`), it is appended to the filteredClients array list. Then, it is displayed to the user along with options to manage the client.

Jumping back to the beginning, when a new instance of Search is created, the following constructor runs:


Search.java

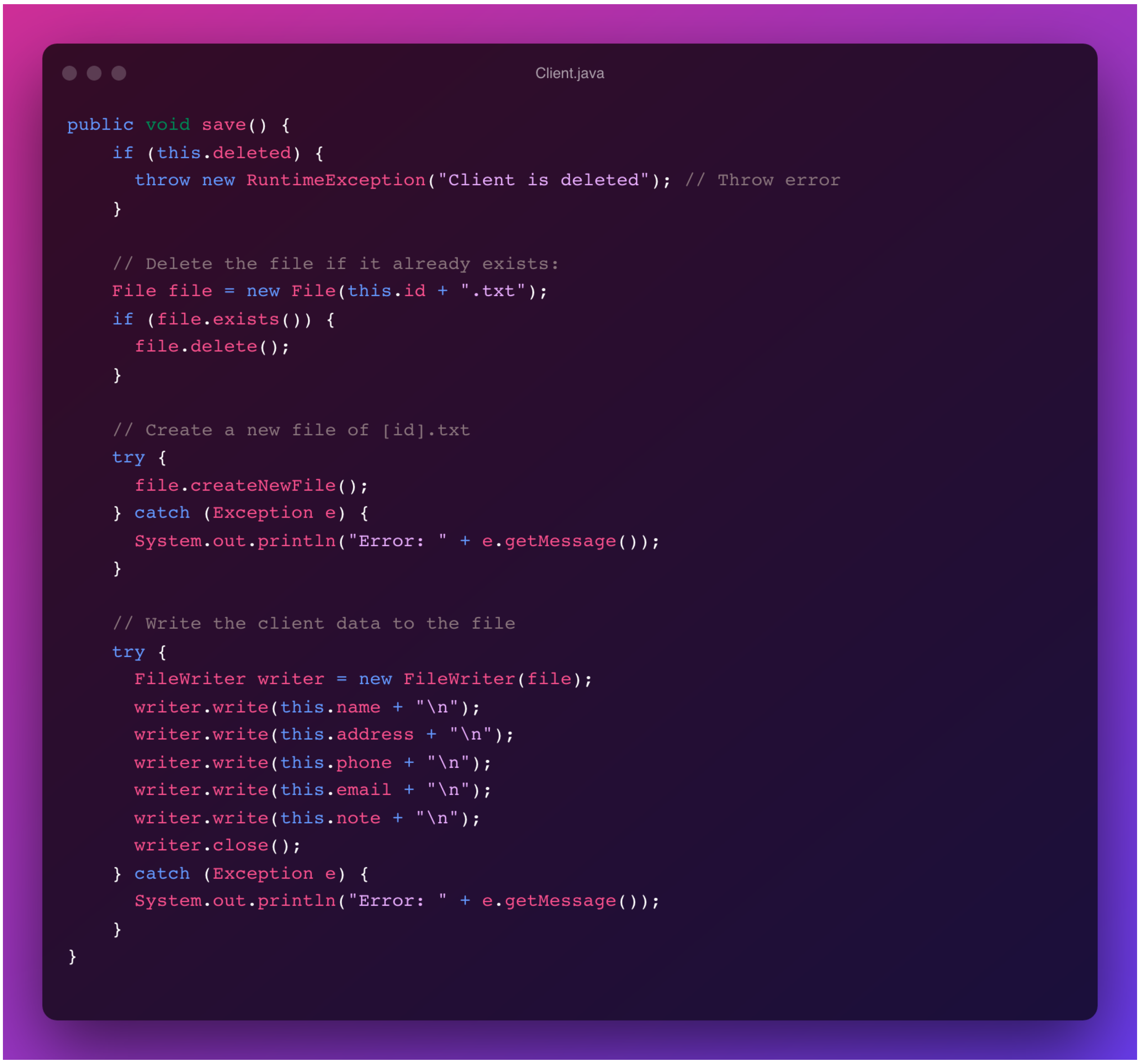
```java
public Search() {
    // Get list of all .txt files in default directory
    File[] files = new File(".").listFiles(new FilenameFilter() {
      public boolean accept(File dir, String name) {
        return name.toLowerCase().endsWith(".txt");
      }
    });

    // Reverse files array to simulate a stack
    for (int i = files.length - 1; i >= 0; i--) {
      // Get file name without extension
      File file = files[i];
      String id = file.getName().replace(".txt", "");
      // Make into an instance of Client
      Client client = new Client(id);
      clients.add(client);

      // Remove file from array to simulate stack
      files[i] = null;
    }
}
```

Firstly, an array containing type File is created with a list of files using a FilenameFilter that accepts .txt files only[1]. This array is intended to be a stack and performs as a stack. Another way of modelling first-in-first-out is looping backwards through the array. For each file, an instance of Client is created and added to the clients ArrayList. Finally, to simulate the pop() functionality of a stack the files[i] reference is set to null.

This shows algorithmic thinking by leveraging multiple data structures, loops, and more to create a powerful layered search feature.

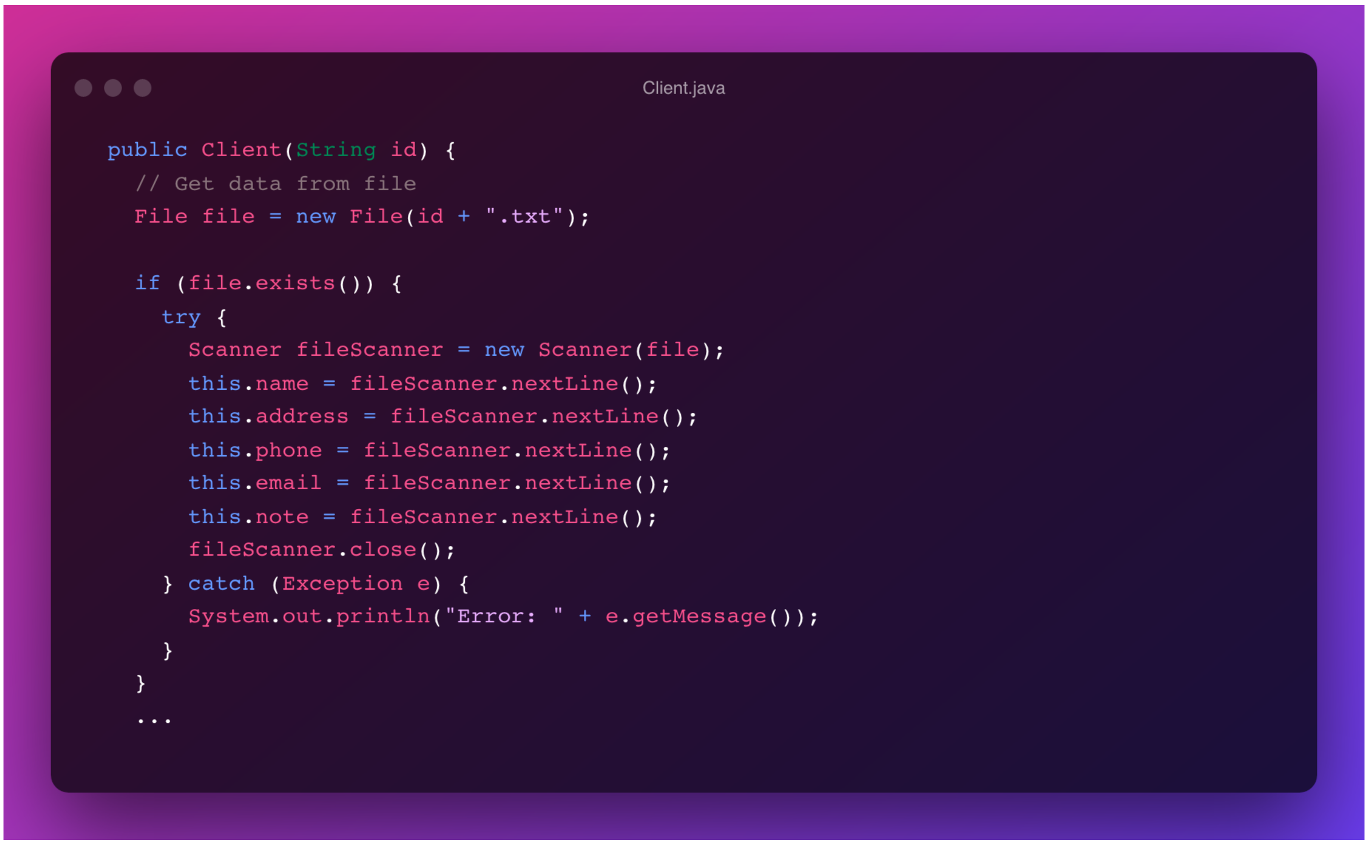
Persistent Data Management

A very important requirement for the client is to ensure the data is persistent. This means that the data is preserved between system restarts. In order to fulfil this requirement, the data must be stored on a persistent data management system such as a disk, either locally or in the cloud (ie. Firebase). To keep the program quick and to preserve privacy, storing data locally is ideal. Conveniently, Java provides useful libraries and methods for this:



```java
public void save() {
    if (this.deleted) {
      throw new RuntimeException("Client is deleted"); // Throw error
    }

    // Delete the file if it already exists:
    File file = new File(this.id + ".txt");
    if (file.exists()) {
      file.delete();
    }

    // Create a new file of [id].txt
    try {
      file.createNewFile();
    } catch (Exception e) {
      System.out.println("Error: " + e.getMessage());
    }

    // Write the client data to the file
    try {
      FileWriter writer = new FileWriter(file);
      writer.write(this.name + "\n");
      writer.write(this.address + "\n");
      writer.write(this.phone + "\n");
      writer.write(this.email + "\n");
      writer.write(this.note + "\n");
      writer.close();
    } catch (Exception e) {
      System.out.println("Error: " + e.getMessage());
    }
}
```

Pictured above is the encapsulated save method for saving client data to disk. Firstly, a `RuntimeException` is thrown if the client is already deleted. Next, object `file` of type `File` is created which contains the file that stores the client's data. If it exists, it is deleted because the data to be replace it is more recent. Finally, a `FileWriter` is created to save the client data to the file. This is wrapped in a try-catch block to ensure proper error handling.


Client.java

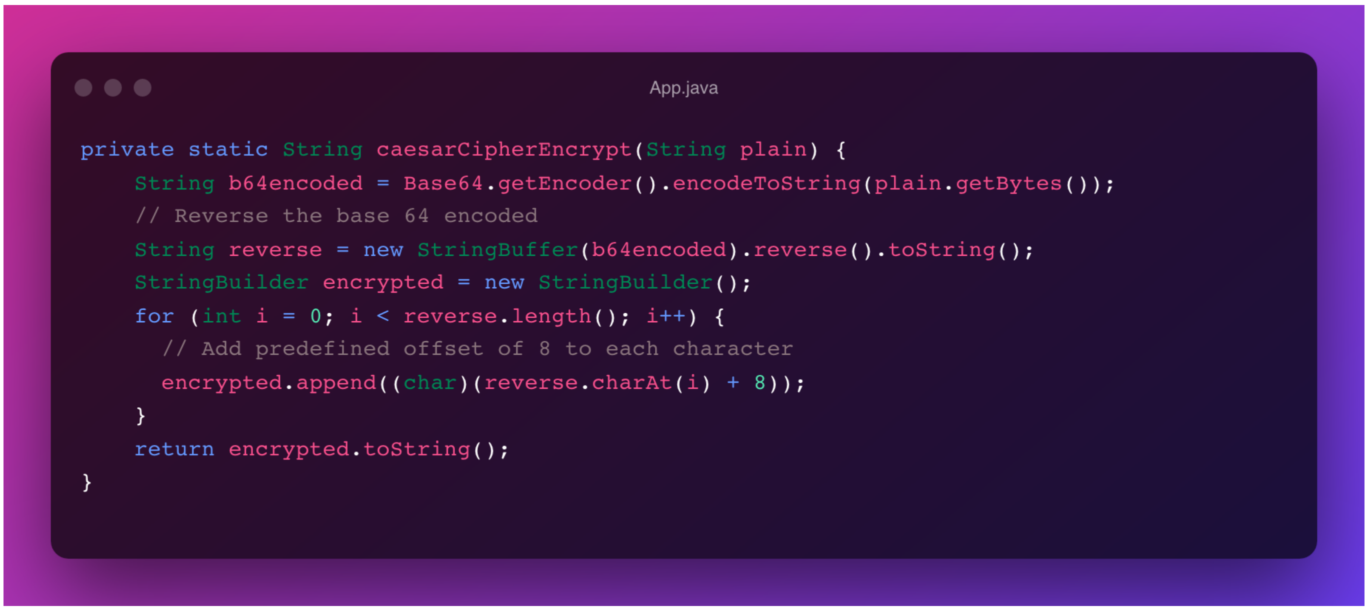
```java
public Client(String id) {
  // Get data from file
  File file = new File(id + ".txt");

  if (file.exists()) {
    try {
      Scanner fileScanner = new Scanner(file);
      this.name = fileScanner.nextLine();
      this.address = fileScanner.nextLine();
      this.phone = fileScanner.nextLine();
      this.email = fileScanner.nextLine();
      this.note = fileScanner.nextLine();
      fileScanner.close();
    } catch (Exception e) {
      System.out.println("Error: " + e.getMessage());
    }
  }
  ...
```

On initialization of an instance of Client (in the constructor method), if the file exists, the file is read using `FileScanner`, and the data is moved into the respective instance fields. Then, importantly, `fileScanner.close()` is called to avoid a memory/data leak. Once again, a try-catch block is used in the event `fileScanner` throws an exception.

This shows algorithmic thinking by leveraging file scanning, error handling, and more to efficiency and effectively read/write relevant data to the disk in a modular fashion.

Caesar Cipher Encryption

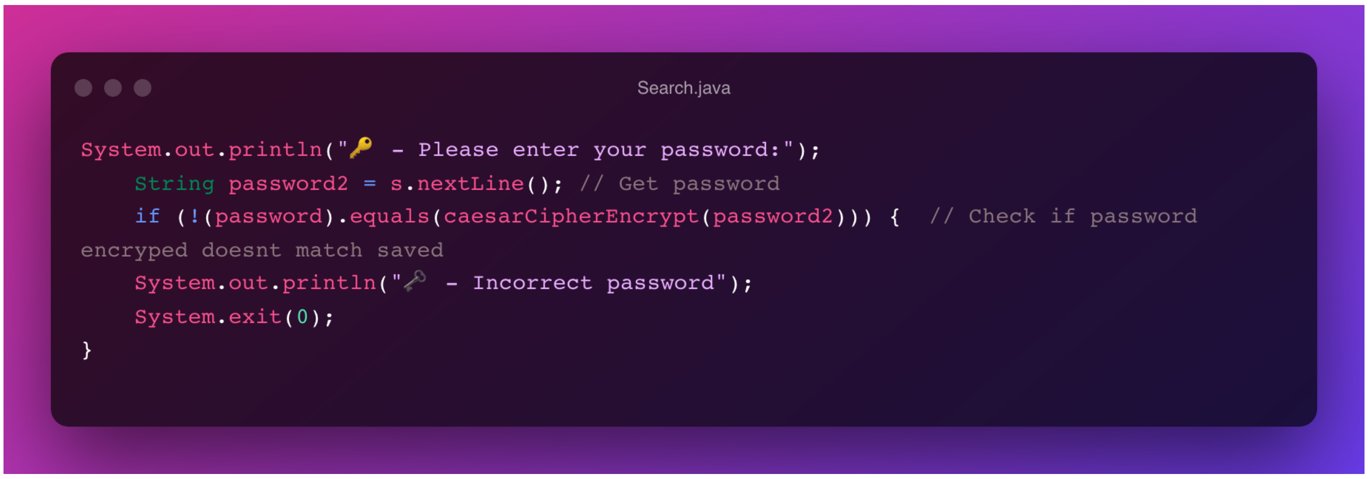
It is a common shortcoming of software is storing users' passwords in plain text on the disk. The reason why is that any malware or bad actor can easily view the password since it's not encrypted. In order to combat this, I created a simple encryption algorithm based on the common caesar cipher.


App.java

```java
private static String caesarCipherEncrypt(String plain) {
    String b64encoded = Base64.getEncoder().encodeToString(plain.getBytes());
    // Reverse the base 64 encoded
    String reverse = new StringBuffer(b64encoded).reverse().toString();
    StringBuilder encrypted = new StringBuilder();
    for (int i = 0; i < reverse.length(); i++) {
      // Add predefined offset of 8 to each character
      encrypted.append((char)(reverse.charAt(i) + 8));
    }
    return encrypted.toString();
}
```

Firstly, using the built-in Java `base64` library, the bytes of the password (`plain`) is encoded using the base64 algorithm. Then, using `StringBuffer`, the base464 encoded string is reversed. Finally, each character of the reversed string is looped through in order to build a new string with `StringBuilder` such that each character is offset by 8. This works because characters in Java have an integer equivalent.

In order to validate a password against this, the following is used:



```java
System.out.println("🔑 - Please enter your password:");
    String password2 = s.nextLine(); // Get password
    if (!(password).equals(caesarCipherEncrypt(password2))) {  // Check if password
encryped doesnt match saved
    System.out.println("🗝 - Incorrect password");
    System.exit(0);
}
```

Since using the same seed phrase (password) always results in the same final encrypted value, simply encrypting a proposed password and comparing it with the encrypted form of the existing password is a secure way to validate the password. This way, there is no need to ever decrypt the password which may expose security vulnerabilities. There are many limitations to this however. For example, would could simply read the files from disk directly rather than using the program. While imperfect, and relatively simple to crack, the caesar cipher is a good proof of concept for making local data more secure.

This shows algorithmic thinking by using a series of methods to scramble a passphrase. Further, securely verifying the password against the saved string.

[1]: WhiteFang34. (2011, April 22). Using File.listFiles with FileNameExtensionFilter. Stack Overflow. Retrieved June 3, 2022, from https://stackoverflow.com/questions/5751335/using-file-listfiles-with-filenameextensionfilter/5751357#5751357.