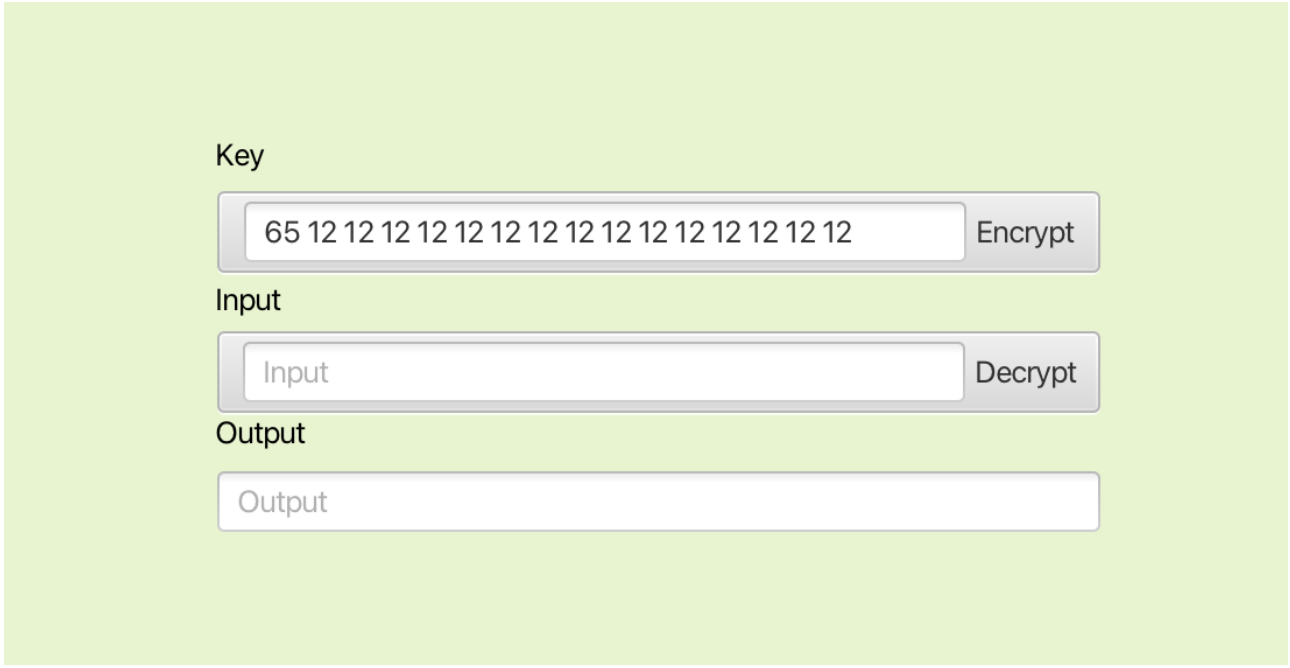
JavaFX Studies/ Aula10: Encryption e Hashing

Primeiro criamos uma tela para demonstrar o AES Encryption, uma tela com uma key de bits, sendo eles de 128 até 256 bits, com essa chave posso colocar oq eu quero encriptar na minha label de input, e clicar para encriptar, assim ele me retorna um resultado daquele meu valor do input encriptado, dessa forma a única maneira de tirar essa encriptation é colocando essa encriptation no meu text field input e mantendo a mesma key e apertando o Decrypt:



Primeiro vamos explicar que necessitamos de um model com uma classe separada chamada Encryptor:

1. **Vetor de Inicialização (IV):**

```
   static byte[] IV = { 0x01, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
0x0c, 0x0d, 0x0e, 0x0f };
```

   - Define um vetor de inicialização estático de 16 bytes utilizado no modo de operação AES CBC.

```
    public String encrypt(String input, byte[] secretKey) throws NoSuchPaddingException,
NoSuchAlgorithmException,
            InvalidAlgorithmParameterException, InvalidKeyException,
            BadPaddingException, IllegalBlockSizeException {
```

2. **Método `encrypt`:**

   - Declaração do método `encrypt` que aceita uma string de entrada e uma chave como parâmetros e retorna uma string criptografada.

```
       Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
```

   - Obtém uma instância do objeto `Cipher` utilizando o algoritmo AES no modo CBC (Cipher Block Chaining) com preenchimento PKCS5.

```
        SecretKeySpec key = new SecretKeySpec(secretKey, "AES");
```

- Converte a chave fornecida para um objeto `SecretKeySpec` apropriado para o algoritmo AES.

```
        cipher.init(Cipher.ENCRYPT_MODE, key, new IvParameterSpec(IV));
```

- Inicializa o objeto `Cipher` para o modo de criptografia usando a chave e o vetor de inicialização.

```
        byte[] cipherText = cipher.doFinal(input.getBytes());
```

- Converte a string de entrada para bytes e criptografa utilizando o objeto `Cipher`.

```
        return Base64.getEncoder().encodeToString(cipherText);
```

- Codifica o texto cifrado em Base64 e retorna o resultado.

```
    public String decrypt(String cipherText, byte[] secretKey) throws NoSuchPaddingException,
NoSuchAlgorithmException,
            InvalidAlgorithmParameterException, InvalidKeyException,
            BadPaddingException, IllegalBlockSizeException {
```

3. **Método `decrypt`:**
   - Declaração do método `decrypt` que aceita uma string cifrada e uma chave como parâmetros e retorna a string descriptografada.

   - A estrutura e o funcionamento desse método são semelhantes ao método `encrypt`, mas com a diferença de que o objeto `Cipher` é inicializado no modo de descriptografia (`Cipher.DECRYPT_MODE`).

```
    public byte[] stringToByteArray(String keyString) {
        String[] keyFragments = keyString.split(" ");

        byte[] key = new byte[16];
        for (int i = 0; i < keyFragments.length; i++) {
            key[i] = Byte.parseByte(keyFragments[i]);
        }
        return key;
    }
```

4. **Método `stringToByteArray`:**
   - Converte uma string de chave formatada em bytes.

```
        String[] keyFragments = keyString.split(" ");
```

   - Divide a string de chave em fragmentos usando um espaço como delimitador.

```
        byte[] key = new byte[16];
        for (int i = 0; i < keyFragments.length; i++) {
            key[i] = Byte.parseByte(keyFragments[i]);
        }
```

   - Converte cada fragmento da chave para byte e preenche um array de bytes.

```
    public static void main(String[] args) throws NoSuchPaddingException, InvalidKeyException,
NoSuchAlgorithmException, IllegalBlockSizeException, BadPaddingException,
InvalidAlgorithmParameterException {
        Encryptor encryptor = new Encryptor();
```

```
        // 128 bit
        byte[] encryptionKey = {65, 12, 12, 12, 12, 12, 12, 12, 12,
                12, 12, 12, 12, 12, 12, 12 };

        String stringKey = "65 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12";

        byte[] key = encryptor.stringToByteArray(stringKey);

        String input = "Secret";

        System.out.println(encryptor.encrypt(input,key));
        // output: VyEcl0pLeqQLemGONcik0w==

        System.out.println(encryptor.decrypt("VyEcl0pLeqQLemGONcik0w==",key));
    }
```

5. **Método `main`:**
   - Cria uma instância da classe `Encryptor`.

```
        byte[] encryptionKey = {65, 12, 12, 12, 12, 12, 12, 12, 12,
                12, 12, 12, 12, 12, 12, 12 };
```

   - Define uma chave de criptografia como um array de bytes.

```
        String stringKey = "65 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12";
```

   - Define a mesma chave como uma string.

```
        byte[] key = encryptor.stringToByteArray(string
```

Essencialmente, o método main serve como um exemplo de como usar a classe Encryptor para criptografar e descriptografar uma mensagem. Ele utiliza uma chave específica e exibe os resultados no console. Este exemplo é uma simulação, e em um ambiente real, você usaria chaves e mensagens mais seguras, além de implementar práticas adequadas de segurança.

Agora vamos explicar o Controller dessa aplicação:

Claro, agora vou explicar detalhadamente cada parte da classe `Controller01`:

1.**
```
public class Controller01 {

    Encryptor encryptor = new Encryptor();
```

2. **Declaração da Classe:**

```
   public class Controller01 {
```

   - Declaração da classe `Controller01`.

```
    Encryptor encryptor = new Encryptor();
```

   - Cria uma instância da classe `Encryptor` para poder usar seus métodos de criptografia e descriptografia.

```
    @FXML
    private TextField keyTextField;
```

```
    @FXML
    private TextField inputTextField;

    @FXML
    private TextField outputTextField;
```

3. **Campos da Interface Gráfica (FXML):**

```
   @FXML
   private TextField keyTextField;

   @FXML
   private TextField inputTextField;

   @FXML
   private TextField outputTextField;
```

   - Anotações `@FXML` indicam que esses campos são injetados a partir do arquivo FXML associado. Eles representam elementos de entrada e saída na interface gráfica (caixas de texto).

```
    @FXML
    void decryptButton(ActionEvent event) throws NoSuchPaddingException, InvalidKeyException, NoSuchAlgorithmException, IllegalBlockSizeException, BadPaddingException, InvalidAlgorithmParameterException {
        // Could be 128, 192, or 256 bits.
        byte[] key = encryptor.stringToByteArray(keyTextField.getText());
        String input = inputTextField.getText();
        String encryptedString = encryptor.decrypt(input, key);
        outputTextField.setText(encryptedString);
    }
```

4. **Método `decryptButton`:**

```
   @FXML
   void decryptButton(ActionEvent event) throws NoSuchPaddingException, InvalidKeyException, NoSuchAlgorithmException, IllegalBlockSizeException, BadPaddingException, InvalidAlgorithmParameterException {
       // ... (Veja explicação abaixo)
   }
```

   - Método associado ao evento do botão de descriptografia na interface gráfica.

```
       byte[] key = encryptor.stringToByteArray(keyTextField.getText());
```

   - Obtém a chave da caixa de texto na interface gráfica e converte para um array de bytes.

```
       String input = inputTextField.getText();
```

   - Obtém a mensagem cifrada da caixa de texto na interface gráfica.

```
       String encryptedString = encryptor.decrypt(input, key);
```

   - Chama o método `decrypt` da instância `Encryptor` para descriptografar a mensagem usando a chave.

```
       outputTextField.setText(encryptedString);
```

- Define o resultado descriptografado no campo de saída da interface gráfica.

```
    @FXML
    void encryptButton(ActionEvent event) throws NoSuchPaddingException, InvalidKeyException,
NoSuchAlgorithmException, IllegalBlockSizeException, BadPaddingException,
InvalidAlgorithmParameterException {
        // Could be 128, 192, or 256 bits.
        byte[] key = encryptor.stringToByteArray(keyTextField.getText());
        String input = inputTextField.getText();

        String encryptedString = encryptor.encrypt(input, key);
        outputTextField.setText(encryptedString);
    }
```

5. **Método `encryptButton`:**

```
    @FXML
    void encryptButton(ActionEvent event) throws NoSuchPaddingException, InvalidKeyException,
NoSuchAlgorithmException, IllegalBlockSizeException, BadPaddingException,
InvalidAlgorithmParameterException {
        // ... (Veja explicação abaixo)
    }
```

- Método associado ao evento do botão de criptografia na interface gráfica.

```
        byte[] key = encryptor.stringToByteArray(keyTextField.getText());
```

- Obtém a chave da caixa de texto na interface gráfica e converte para um array de bytes.

```
        String input = inputTextField.getText();
```

- Obtém a mensagem original da caixa de texto na interface gráfica.

```
        String encryptedString = encryptor.encrypt(input, key);
```

- Chama o método `encrypt` da instância `Encryptor` para criptografar a mensagem usando a chave.

```
        outputTextField.setText(encryptedString);
```

- Define o resultado criptografado no campo de saída da interface gráfica.

Esse controlador está vinculado a uma interface gráfica (FXML) e fornece métodos para lidar com eventos de botões, como criptografar e descriptografar mensagens usando a classe `Encryptor`. Ele faz uso de campos de texto na interface para entrada e exibição de dados. As operações são acionadas pelos eventos associados aos botões na interface gráfica.

Vou explicar agora a metodologia AES (Advanced Encryption Standard) e como ela é utilizada na classe `Encryptor` junto com a funcionalidade da classe `Cipher`.

AES (Advanced Encryption Standard):

O AES é um algoritmo de criptografia simétrica que substituiu o DES (Data Encryption Standard) como o padrão de criptografia para o governo dos Estados Unidos. Ele opera em blocos de dados e suporta chaves de 128, 192 e 256 bits.

Modo de Operação - CBC (Cipher Block Chaining):

O modo de operação CBC é utilizado na classe `Encryptor`. Neste modo, cada bloco de texto simples é XOR (OU exclusivo) com o bloco de texto cifrado anterior antes da criptografia. Além

disso, um vetor de inicialização (IV) é usado como entrada no primeiro bloco. Isso garante que blocos idênticos de texto simples não gerem blocos idênticos de texto cifrado.

Funcionalidade da Classe `Cipher`:

A classe `Cipher` é parte do pacote `javax.crypto` e fornece a implementação de algoritmos criptográficos. Neste contexto, ela é utilizada para realizar operações de criptografia e descriptografia usando o AES.

Métodos Principais Utilizados:

1. **`Cipher.getInstance("AES/CBC/PKCS5Padding")`:**

   - Obtém uma instância do objeto `Cipher` com o algoritmo AES, modo CBC e preenchimento PKCS5. Este método é usado para definir as configurações de criptografia.

2. **`cipher.init(Cipher.ENCRYPT_MODE, key, new IvParameterSpec(IV))`:**

   - Inicializa o objeto `Cipher` para o modo de criptografia (ou descriptografia) com a chave e o vetor de inicialização especificados.

3. **`cipher.doFinal(input.getBytes())`:**

   - Executa a operação de criptografia ou descriptografia nos dados fornecidos. No caso da criptografia, converte a string de entrada para bytes e retorna o texto cifrado.

4. **`Base64.getEncoder().encodeToString(cipherText)`:**

   - Converte os bytes do texto cifrado para uma representação em Base64, facilitando a manipulação e transmissão dos dados.

5. **`Base64.getDecoder().decode(cipherText)`:**

   - Converte uma string em Base64 de volta para os bytes originais antes de realizar a descriptografia.

Fluxo Geral na Classe `Encryptor`:

1. **Iniciando uma Instância do `Cipher`:**

   - Um objeto `Cipher` é criado usando o algoritmo AES, modo CBC e preenchimento PKCS5.

2. **Convertendo a Chave:**

   - A chave fornecida como array de bytes é convertida para um objeto `SecretKeySpec`, que é apropriado para o algoritmo AES.

3. **Inicializando o `Cipher`:**

   - O `Cipher` é inicializado no modo desejado (criptografia ou descriptografia) usando a chave e o vetor de inicialização.

4. **Executando a Operação:**

   - A operação (criptografia ou descriptografia) é realizada nos dados fornecidos.

5. **Codificando/Decodificando em Base64:**

   - O resultado da operação é convertido para Base64 para facilitar a manipulação e transmissão dos dados.

Considerações Finais:

O AES é amplamente utilizado devido à sua segurança e eficiência. O modo CBC é apenas um dos modos de operação disponíveis, e a escolha do modo depende dos requisitos específicos do sistema.

A classe `Cipher` facilita a implementação de algoritmos criptográficos e oferece uma maneira robusta de garantir a confidencialidade e integridade dos dados. É importante manter as chaves seguras e seguir as boas práticas de segurança ao implementar algoritmos criptográficos.

Ultimas considerações e explicações:

A classe Encryptor implementa a criptografia simétrica AES. Aqui estão as principais partes:

- **Inicialização e Chave:**

  Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding"); SecretKeySpec key = new SecretKeySpec(secretKey, "AES"); cipher.init(Cipher.ENCRYPT_MODE, key, new IvParameterSpec(IV));

  - Um objeto Cipher é criado usando o algoritmo AES no modo CBC (Cipher Block Chaining) com preenchimento PKCS5.
  - A chave é convertida em um objeto SecretKeySpec, que é necessário para inicializar o Cipher.
  - O Cipher é inicializado no modo de criptografia (ENCRYPT_MODE) com a chave e o vetor de inicialização (IV).

- **Criptografia:**

  byte[] cipherText = cipher.doFinal(input.getBytes());

  - A mensagem de entrada é convertida para bytes.
  - O método doFinal é chamado para realizar a criptografia.
  - O texto cifrado resultante é armazenado em um array de bytes.

- **Conversão para Base64:**

  return Base64.getEncoder().encodeToString(cipherText);

  - O texto cifrado é convertido para uma representação Base64 para facilitar o armazenamento e transmissão.

- **Descriptografia:**

  cipher.init(Cipher.DECRYPT_MODE, key, new IvParameterSpec(IV)); byte[] plainText = cipher.doFinal(Base64.getDecoder().decode(cipherText));

  - Para descriptografar, um novo objeto Cipher é inicializado no modo de descriptografia (DECRYPT_MODE) usando a mesma chave e IV.
  - O texto cifrado, convertido de Base64, é descriptografado usando o método doFinal.
  - O resultado é um array de bytes representando a mensagem original.

Funcionalidade da Classe Cipher**:**
A classe Cipher no Java é uma parte importante para realizar operações criptográficas. Ela oferece uma interface para vários algoritmos criptográficos, incluindo AES. Aqui estão algumas características-chave:

- Cipher.getInstance("AES/CBC/PKCS5Padding"):
  - Especifica o algoritmo de criptografia (AES) e o modo de operação (CBC) com o esquema de preenchimento PKCS5.

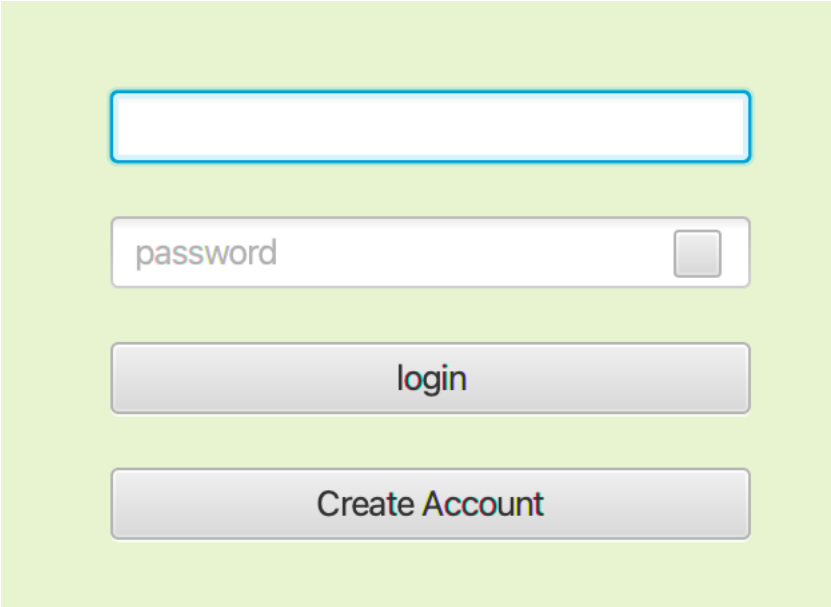
- cipher.init(...):
  - Inicializa o objeto Cipher para criptografia ou descriptografia.
  - Requer uma chave, um modo (ENCRYPT_MODE ou DECRYPT_MODE) e, no caso do modo CBC, um vetor de inicialização (IV).

- cipher.doFinal(...) **e** cipher.update(...):
  - Métodos que realizam as operações criptográficas. doFinal é geralmente usado para processar a última parte dos dados e finalizar a operação.

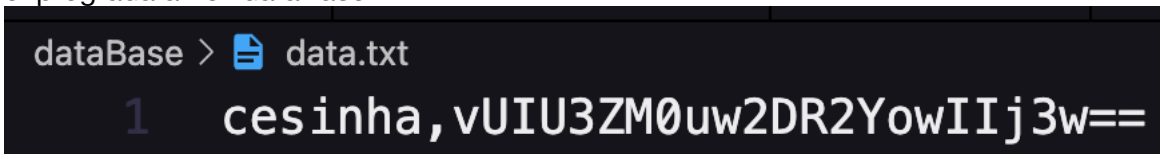
- SecretKeySpec:
  - Uma classe que encapsula a chave como uma implementação específica do algoritmo. Neste caso, para o AES.

- IvParameterSpec:
  - Uma classe que especifica um vetor de inicialização (IV) necessário para alguns modos de operação, como o CBC.

Agora vamos ver uma aplicação em FX de um sistema de login que usa o AES Encryptation:



Esse programa utiliza a classe Encryptor igualmente acima pra criptografar as senhas dos usuários no bancodedados, nesse caso quando colocamos credenciais ai e clicamos em create account nós criamos um banco de dados improvisado por meio de um arquivo.txt chamado data.txt e esse arquivo armazena o login e a senha do usuário, porém a senha do usuário não pode ficar visível para o usuário, para isso essa senha para pelo método de encryptor e aparece criptogradafa no “dataBase”:



Agora vamos explicar o Controller dessa classe:

Vamos analisar detalhadamente as seções restantes da classe Controller02, focando nas operações de login, escrita e leitura do arquivo dataBase/data.txt:

@FXML void loginHandler(ActionEvent event) throws IOException, NoSuchPaddingException, InvalidKeyException, NoSuchAlgorithmException, IllegalBlockSizeException, BadPaddingException, InvalidAlgorithmParameterException { String username = usernameTextField.getText(); String password = getPassword(); updateUsernamesAndPasswords(); String encryptedPassword = loginInfo.get(username); if (password.equals(encryptor.decrypt(encryptedPassword, encryptionKey))) { System.out.println("successfully login!"); } else { errorField.setVisible(true); } }

- **Método** loginHandler**:**

  @FXML void loginHandler(ActionEvent event) throws IOException, NoSuchPaddingException, InvalidKeyException, NoSuchAlgorithmException, IllegalBlockSizeException, BadPaddingException, InvalidAlgorithmParameterException { // ... (Veja explicação abaixo) }

- Método associado ao evento de clique no botão de login.
  String username = usernameTextField.getText(); String password = getPassword(); updateUsernamesAndPasswords();

  - Obtém o nome de usuário e a senha fornecidos pelo usuário.
  - Chama o método updateUsernamesAndPasswords para garantir que as informações de login estejam atualizadas.
    String encryptedPassword = loginInfo.get(username);
  - Obtém a senha criptografada associada ao nome de usuário fornecido.
    if (password.equals(encryptor.decrypt(encryptedPassword, encryptionKey))) { System.out.println("successfully login!"); } else { errorField.setVisible(true); }
  - Compara a senha fornecida (descriptografada) com a senha armazenada.
  - Se as senhas coincidirem, exibe uma mensagem no console indicando um login bem-sucedido.
  - Caso contrário, torna visível um campo de texto de erro na interface gráfica.

@FXML void createAccount(ActionEvent event) throws IOException, NoSuchPaddingException, InvalidAlgorithmParameterException, NoSuchAlgorithmException, IllegalBlockSizeException, BadPaddingException, InvalidKeyException { writeToFile(); }

- **Método** createAccount**:**

  @FXML void createAccount(ActionEvent event) throws IOException, NoSuchPaddingException, InvalidAlgorithmParameterException, NoSuchAlgorithmException, IllegalBlockSizeException, BadPaddingException, InvalidKeyException { // ... (Veja explicação abaixo) }

  - Método associado ao evento de criação de uma nova conta.
    writeToFile();
  - Chama o método writeToFile para escrever as informações do novo usuário no arquivo.

private String getPassword(){ if(passwordTextField.isVisible()){ return passwordTextField.getText(); } else { return hiddenPasswordTextField.getText(); } }

- **Método** getPassword**:**
  private String getPassword(){ // ... (Veja explicação abaixo) }

  - Método que retorna a senha com base na visibilidade dos campos de senha na interface gráfica.
    if(passwordTextField.isVisible()){ return passwordTextField.getText(); } else { return hiddenPasswordTextField.getText(); }

  - Se a caixa de senha visível estiver sendo exibida, retorna o texto dessa caixa.
  - Caso contrário, retorna o texto da caixa de senha oculta.

private void updateUsernamesAndPasswords() throws IOException { Scanner scanner = new Scanner(file); loginInfo.clear(); while (scanner.hasNext()){ String[] splitInfo = scanner.nextLine().split(","); loginInfo.put(splitInfo[0],splitInfo[1]); } scanner.close(); }

- **Método** updateUsernamesAndPasswords**:**

  private void updateUsernamesAndPasswords() throws IOException { // ... (Veja explicação abaixo) }

  - Método que lê as informações de login do arquivo e atualiza o HashMap loginInfo:
    Scanner scanner = new Scanner(file); loginInfo.clear(); while (scanner.hasNext()){ String[] splitInfo = scanner.nextLine().split(","); loginInfo.put(splitInfo[0],splitInfo[1]); } scanner.close();

  - Cria um scanner para ler o conteúdo do arquivo linha por linha.
  - Limpa o HashMap para evitar duplicatas.
  - Para cada linha do arquivo, divide as informações de login (username e senha) usando a vírgula como delimitador e as adiciona ao HashMap.

private void writeToFile() throws IOException, NoSuchPaddingException, InvalidKeyException, NoSuchAlgorithmException, IllegalBlockSizeException, BadPaddingException, InvalidAlgorithmParameterException { String username = usernameTextField.getText(); String password = getPassword(); BufferedWriter writer = new BufferedWriter(new FileWriter(file,true)); writer.write(username + "," + encryptor.encrypt

O pulo de linha (quebra de linha) no arquivo é alcançado através do método write do objeto BufferedWriter. Vou explicar essa parte do código em detalhes:

writer.write(username + "," + encryptor.encrypt(password, encryptionKey) + "\n");
  - writer.write(username + "," + encryptor.encrypt(password, encryptionKey) + "\n");**:**
  - Este comando escreve uma nova linha no arquivo. A parte crucial é o "\n", que representa um caractere de nova linha (LF - Line Feed).
  - O método write escreve a string username + "," + encryptor.encrypt(password, encryptionKey) no arquivo, seguida por uma nova linha.
  - A expressão username + "," + encryptor.encrypt(password, encryptionKey) forma a linha no arquivo, onde o nome de usuário e a senha criptografada estão separados por uma vírgula.

Assim, sempre que um novo usuário é adicionado ao arquivo, ele é colocado em uma nova linha, permitindo a leitura e o processamento adequados das informações durante a leitura posterior do arquivo.

A leitura e interpretação das quebras de linha dependem do método de leitura e processamento utilizado. No código fornecido, a leitura das linhas do arquivo é feita utilizando um Scanner da seguinte maneira:

private void updateUsernamesAndPasswords() throws IOException { Scanner scanner = new Scanner(file); loginInfo.clear(); while (scanner.hasNext()){ String[] splitInfo = scanner.nextLine().split(","); loginInfo.put(splitInfo[0], splitInfo[1]); } scanner.close(); }

Aqui estão os pontos chave:

- **Scanner.nextLine():**
  - O método nextLine() lê a linha inteira do arquivo, incluindo a quebra de linha.
  - O método nextLine() retorna uma string contendo todos os caracteres da linha, incluindo a quebra de linha no final.
- **String[] splitInfo = scanner.nextLine().split(",");:**
  - O código usa split(",") para dividir a linha em partes usando a vírgula como delimitador.
  - Portanto, mesmo que haja uma quebra de linha no final da linha, ela será incluída na última parte da string resultante após a divisão.
- **loginInfo.put(splitInfo[0], splitInfo[1]);:**
  - As partes resultantes após a divisão são então usadas para atualizar o HashMap loginInfo.

Isso significa que a quebra de linha não interfere na leitura dos dados. O Scanner lê a linha inteira, incluindo a quebra de linha, e o método split é usado para separar os dados na linha. A última parte da string após a divisão conterá a senha criptografada, incluindo a quebra de linha, mas isso não afeta a funcionalidade do programa, pois a quebra de linha é uma parte da string armazenada no HashMap.

Encryption usando TripleDES:

Primeiro vamos citar as diferenças entre o AES e o TripleDES:

O AES (Advanced Encryption Standard) e o TripleDES (Triple Data Encryption Standard) são algoritmos de criptografia simétrica que visam fornecer segurança para dados sensíveis. Aqui estão algumas diferenças e pontos positivos/negativos de cada um:

AES (Advanced Encryption Standard):
Vantagens do AES:

- **Eficiência:**
  - O AES é conhecido por ser eficiente em termos de desempenho, oferecendo uma boa combinação entre segurança e velocidade de processamento.

- **Padrão Internacional:**
  - O AES foi adotado como padrão pelo Instituto Nacional de Padrões e Tecnologia dos EUA (NIST) e é amplamente utilizado em todo o mundo.

- **Segurança Comprovada:**
  - O AES resistiu a uma ampla variedade de ataques criptográficos e é considerado seguro quando usado corretamente com chaves suficientemente longas.

Desvantagens do AES:

Chaves Fixas:

- Em alguns casos, o uso de chaves fixas para criptografia AES pode introduzir vulnerabilidades, especialmente se as chaves forem fracas ou previsíveis.

TripleDES (Triple Data Encryption Standard):
Vantagens do TripleDES:

- **Herança do DES:**
  - O TripleDES é baseado no DES, que foi amplamente estudado e testado ao longo do tempo. A herança do DES confere alguma confiança na robustez do algoritmo.

- **Flexibilidade de Chaves:**
  - O TripleDES oferece flexibilidade em relação ao tamanho da chave. Pode ser usado com chaves de 128, 192 ou 256 bits, oferecendo opções para diferentes requisitos de segurança.

Desvantagens do TripleDES:

- **Desempenho Inferior ao AES:**
  - Comparado ao AES, o TripleDES é geralmente mais lento em termos de desempenho, especialmente quando se trata de criptografia em hardware.

- **Blocos de Tamanho Pequeno:**
  - O TripleDES opera em blocos de tamanho fixo de 64 bits, o que pode ser considerado um ponto fraco em termos de segurança e flexibilidade.

- **Susceptível a Ataques Diferenciais:**
  - Embora mais robusto que o DES original, o TripleDES ainda pode ser vulnerável a certos tipos de ataques, como ataques diferenciais.

Escolha entre AES e TripleDES:

- **Segurança:**
  - Se segurança é a prioridade absoluta, o AES é geralmente preferido. Ele oferece uma segurança forte e é amplamente adotado como padrão.

- **Desempenho:**
  - Se o desempenho é crítico, especialmente em ambientes com recursos limitados, o AES é frequentemente a escolha preferida devido à sua eficiência.

- **Compatibilidade:**
  - Em alguns casos, a compatibilidade com sistemas mais antigos pode ser uma consideração, e o TripleDES pode ser uma escolha válida.

Em resumo, a escolha entre AES e TripleDES depende dos requisitos específicos do sistema, como nível de segurança desejado, desempenho e compatibilidade com sistemas existentes. Em muitos casos, especialmente em novos desenvolvimentos, o AES é a escolha preferida devido à sua segurança comprovada e eficiência.

Primeiramente em TripleDES podemos ter chaves em formato de Strings ao invés de 16 números hexadecimal, vamos explicar o código do EncryptorTripleDES no model da nossa aplicação e suas diferenças comparadas ao nosso EncryptorAES:

Método `encrypt`:

```
public String encrypt(String input, String keyString) throws NoSuchAlgorithmException,
UnsupportedEncodingException, NoSuchPaddingException,
InvalidAlgorithmParameterException, InvalidKeyException, BadPaddingException,
IllegalBlockSizeException {
    // ...
```

1. **Geração da Chave:**

   - O método utiliza o algoritmo de hash MD5 para derivar uma chave de 24 bytes a partir da string de chave fornecida.
   - Os primeiros 16 bytes do hash MD5 são copiados para formar a chave, e os últimos 8 bytes são duplicados para completar os 24 bytes necessários para o TripleDES.

2. **Inicialização do Cipher:**

   - Usa um vetor de inicialização (IV) de 8 bytes, inicializado com zeros.
   - Inicializa o objeto `Cipher` no modo de criptografia (`Cipher.ENCRYPT_MODE`) usando a chave derivada e o IV.

3. **Criptografia do Texto Plano:**

   - Converte a string de entrada para bytes.
   - Utiliza o método `doFinal` do `Cipher` para criptografar os bytes do texto plano.

4. **Conversão do Texto Criptografado para String:**

   - Converte os bytes do texto criptografado para uma string, onde os valores de byte são separados por espaços.

Método `decrypt`:

```
public String decrypt(String ciphertext, String keyString)
        throws InvalidKeyException, NoSuchPaddingException,
        InvalidAlgorithmParameterException, IllegalBlockSizeException,
        BadPaddingException, NoSuchAlgorithmException {
    // ...
}
```

1. **Conversão da String Criptografada para Bytes:**

   - Utiliza o método `stringToByteArray` para converter a string criptografada de volta para um array de bytes.

2. **Geração da Chave (Mesmo Processo que no `encrypt`):**

   - Gera a chave usando o mesmo processo descrito no método `encrypt`.

3. **Inicialização do Cipher (Mesmo Processo que no `encrypt`):**

   - Inicializa o objeto `Cipher` no modo de descriptografia (`Cipher.DECRYPT_MODE`) usando a chave derivada e o IV.

4. **Descriptografia do Texto Criptografado:**

   - Utiliza o método `doFinal` do `Cipher` para descriptografar os bytes do texto criptografado.

5. **Conversão dos Bytes Descriptografados para String:**

   - Converte os bytes descriptografados de volta para uma string.

Método `stringToByteArray`:

```
private byte[] stringToByteArray(String string){
    // ...
}
```

Converte uma string de valores de bytes (separados por espaços) de volta para um array de bytes.

Diferenças em relação à Classe `Encryptor` (AES):

1. **Algoritmo Criptográfico:**

   - `EncryptorTripleDES` utiliza o algoritmo TripleDES.
   - `Encryptor` utiliza o algoritmo AES.

2. **Tamanho da Chave:**

   - `EncryptorTripleDES` gera uma chave de 24 bytes (192 bits) para o TripleDES.
   - `Encryptor` utiliza uma chave de 16 bytes (128 bits) para o AES.

3. **Modo de Operação:**

   - Ambos utilizam o modo de operação CBC (Cipher Block Chaining) para garantir a segurança, mas os detalhes podem variar entre os algoritmos.

4. **Geração de Chave a partir da String:**

   - Ambos utilizam um processo de derivação de chave a partir de uma string, porém o AES necessita receber a chave em forma de 16 números hexadecimais enquanto o TripleDES pode receber direto uma String , junto a isso o método exato difere. `Encryptor` utiliza a classe `SecretKeySpec`, enquanto `EncryptorTripleDES` utiliza o algoritmo de hash MD5.

5. **Configuração do Cipher:**

   - As configurações específicas do `Cipher` (como o tamanho do bloco e o preenchimento) podem variar entre AES e TripleDES, refletindo as diferenças nos algoritmos.

6. **Tamanho do Bloco:**

   - AES opera em blocos de 128 bits (16 bytes).
   - TripleDES opera em blocos de 64 bits (8 bytes).

Em resumo, as principais diferenças estão relacionadas ao algoritmo criptográfico, tamanho da chave, modo de operação e detalhes específicos da configuração do `Cipher`. Cada algoritmo tem suas próprias características e considerações de segurança, e a escolha entre eles depende dos requisitos específicos de aplicação e segurança.

MD5:

Primeiro vamos citar a principal diferença entre esse modo MD5 e os anteriores:

- **AES (Advanced Encryption Standard):**
  - O AES é um algoritmo de criptografia simétrica (de chave secreta), e não é uma função de hash. Ele é usado para criptografar e descriptografar dados. O processo é reversível, ou seja, você pode recuperar os dados originais usando a chave correta.

- **TripleDES (Triple Data Encryption Standard):**
  - Semelhante ao AES, o TripleDES é um algoritmo de criptografia simétrica e não envolve funções de hash. Ele é usado para criptografia e descriptografia de dados.

- **MD5 (Message Digest Algorithm 5):**

- O MD5 é uma função de hash criptográfico, não uma técnica de criptografia. Ele é usado para gerar um valor de hash fixo de 128 bits (ou 32 caracteres hexadecimais) a partir de uma entrada. No entanto, MD5 é considerado obsoleto e não seguro para aplicações críticas de segurança devido a vulnerabilidades conhecidas.

Funções de hash, como MD5, são frequentemente usadas em contextos diferentes da criptografia simétrica, como para garantir a integridade de dados, verificar a autenticidade de mensagens e armazenar senhas de maneira segura (mediante o uso de técnicas apropriadas, como o uso de "salt").

Em resumo, AES e TripleDES são algoritmos de criptografia simétrica, enquanto MD5 é uma função de hash. Cada um desempenha um papel específico em diferentes contextos de segurança da informação.

Mas, o que seria um Hash:

Em criptografia e ciência da computação, o termo "hash" refere-se a uma função de hash, que é uma função matemática que mapeia dados de entrada de tamanho arbitrário para um valor de comprimento fixo, geralmente uma sequência de caracteres alfanuméricos. O resultado dessa função é conhecido como "hash" ou "valor de hash".

Aqui estão alguns pontos importantes sobre hashes:

- **Unidirecional (One-Way):**
  - Funções de hash são unidirecionais, o que significa que é fácil calcular o hash de uma entrada, mas é extremamente difícil ou impossível regenerar a entrada original a partir do hash.

- **Tamanho Fixo:**
  - A função de hash gera um valor de tamanho fixo, independentemente do tamanho da entrada. Por exemplo, um algoritmo de hash específico pode gerar hashes de 128 bits, 256 bits ou outro tamanho fixo.

- **Determinístico:**
  - A mesma entrada sempre produzirá o mesmo valor de hash. Isso é crucial para a consistência e utilidade das funções de hash.

- **Rápido de Calcular:**
  - Funções de hash são projetadas para serem eficientes em termos de processamento, permitindo que sejam rapidamente calculadas para grandes volumes de dados.

- **Espalhamento (Avalanche Effect):**
  - Pequenas alterações na entrada devem resultar em mudanças significativas no hash. Isso é conhecido como o efeito avalanche, onde até mesmo uma pequena alteração nos dados de entrada deve resultar em um hash completamente diferente.

- **Colisões:**
  - Duas entradas diferentes podem produzir o mesmo valor de hash. Isso é conhecido como uma colisão. Funções de hash criptográficas são projetadas para minimizar a probabilidade de colisões, tornando-as computacionalmente inviáveis de ocorrer.

- **Uso Comum:**
  - Funções de hash são amplamente utilizadas em várias aplicações, como garantir a integridade de dados (checksums), armazenamento seguro de senhas (hash de senhas), autenticação digital, entre outros.

- **Exemplos Comuns:**
  - Algoritmos de hash comuns incluem MD5, SHA-1 (não mais considerado seguro), SHA-256, SHA-3, entre outros.

As funções de hash desempenham um papel fundamental em muitos aspectos da segurança da informação e ciência da computação, fornecendo uma maneira eficiente e segura de representar dados de forma única e verificável.

Agora voltando ao nosso programa, agora iremos mostrar um sistema de login que utiliza o MD5 como forma de ciptografia de senhas, primeiramente vamos explicar o algoritmo do MD5:

- **Inicialização de Variáveis:**
  - O algoritmo inicia com quatro variáveis de 32 bits, que são inicializadas com valores específicos.

- **Padding da Mensagem:**
  - A mensagem de entrada é preenchida (padded) para garantir que seu comprimento seja um pouco menos de um múltiplo específico de 512 bits. O preenchimento consiste em um bit "1" seguido por zeros e, finalmente, o comprimento original da mensagem é anexado.

- **Processamento por Blocos:**
  - A mensagem é dividida em blocos de 512 bits (64 bytes). Cada bloco é processado individualmente.

- **Operações Bitwise e Funções Não Lineares:**
  - Cada bloco passa por várias operações bitwise (como XOR, AND, OR) e funções não lineares que envolvem operações como shift e adição modular.

- **Atualização das Variáveis de Estado:**
  - As variáveis de estado são continuamente atualizadas após o processamento de cada bloco.

- **Geração do Hash:**
  - Após o processamento de todos os blocos, o valor final dessas variáveis de estado é concatenado para formar o hash final de 128 bits.

Agora vamos explicar o nosso EncryptorMD5 da classe model:

A classe `EncryptorMD5` é uma implementação simples de um "encryptor" (embora o termo correto seria "hasher" no contexto do MD5) que utiliza a função de hash MD5 para gerar um hash a partir de uma string de entrada. Vamos examinar cada parte da classe:

Método `encryptString`:

```
public String encryptString(String input) throws NoSuchAlgorithmException {
    // ...
}
```

1. **Criação do Objeto `MessageDigest`:**

   - Utiliza a classe `MessageDigest` para criar um objeto que implementa o algoritmo de hash MD5.

2. **Obtenção do Digest da Mensagem:**

   - Calcula o digest da mensagem (hash) convertendo a string de entrada para bytes e aplicando a função de hash MD5.

3. **Criação de `BigInteger`:**

   - Converte o array de bytes do hash para um objeto `BigInteger`. O primeiro parâmetro "1" indica que o valor é positivo.

4. **Conversão para Hexadecimal:**

- Converte o valor da `BigInteger` para uma representação hexadecimal. Isso é feito usando `toString(16)`.

5. **Retorno do Hash em Hexadecimal:**

   - Retorna a representação em hexadecimal do hash MD5.

Método `main`:

```
public static void main(String[] args) throws NoSuchAlgorithmException {
    EncryptorMD5 encryptor = new EncryptorMD5();

    // Definição de uma senha e seu hash correspondente
    String password = "monkey123";
    String hashedPas = "cc25c0f861a83f5efadc6e1ba9d1269e";

    Scanner scanner = new Scanner(System.in);

    System.out.println("Hey! Plz input your Password: \n");

    // Leitura da entrada do usuário
    String userInput = scanner.nextLine();

    // Verificação se o hash da entrada do usuário é igual ao hash predefinido
    if (encryptor.encryptString(userInput).equals(hashedPas)) {
        System.out.println("Correct! You are in!");
    } else {
        System.out.println("Wrong!!");
    }
}
```

Funcionamento Geral:

1. O programa cria uma instância da classe `EncryptorMD5`.

2. Define uma senha predefinida (`password`) e seu hash correspondente (`hashedPas`).

3. Solicita que o usuário insira uma senha.

4. Calcula o hash MD5 da senha inserida pelo usuário usando o método `encryptString`.

5. Compara o hash calculado com o hash predefinido e exibe uma mensagem correspondente.

Essa classe ilustra um exemplo básico de uso do MD5 para verificar uma senha. No entanto, é importante observar que o MD5 não é mais considerado seguro para armazenamento seguro de senhas devido a vulnerabilidades conhecidas, e recomenda-se o uso de algoritmos mais robustos, como bcrypt ou Argon2, para essa finalidade. Além disso, é uma prática comum incluir um "salt" ao calcular hashes de senhas para aumentar a segurança.

É importante observar que o MD5 é vulnerável a colisões, o que significa que duas entradas diferentes podem resultar no mesmo valor de hash. Além disso, o MD5 não é mais considerado seguro para aplicações criptográficas devido a vulnerabilidades conhecidas, e algoritmos mais seguros, como **SHA-256 ou SHA-3**, são recomendados para aplicações modernas.

#ID