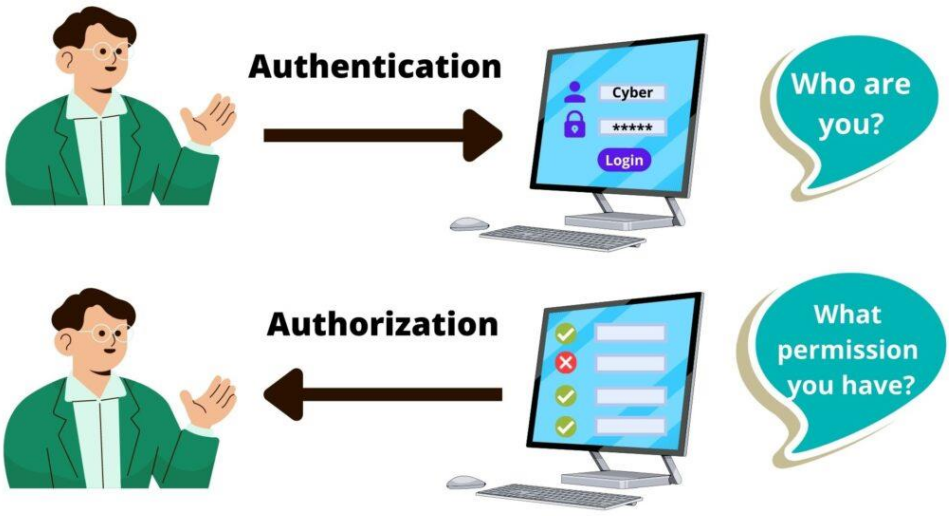
Symmetric Encryption

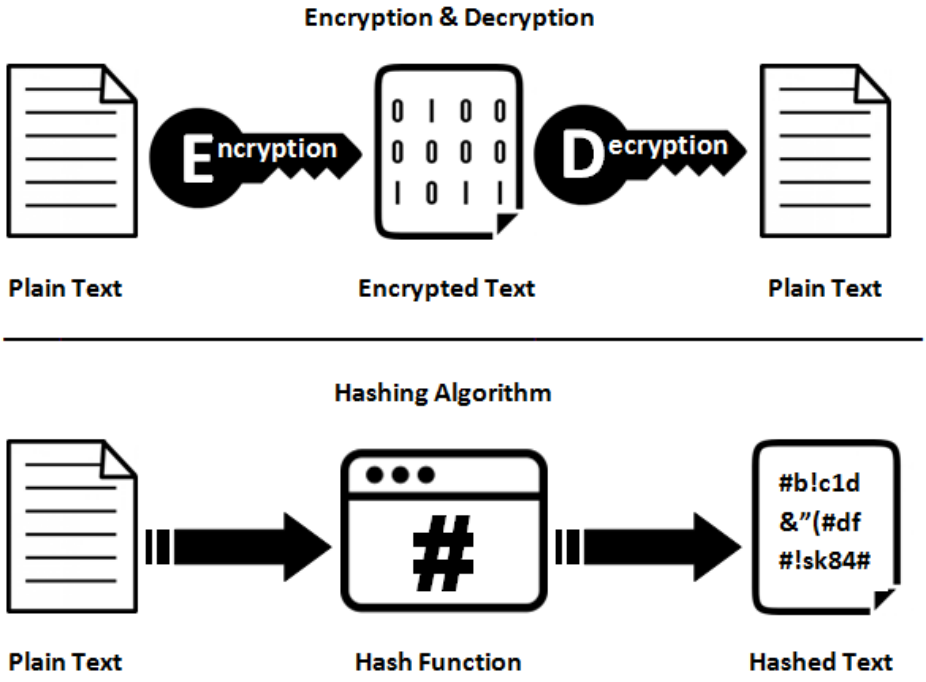
Contents

Basic Terminology

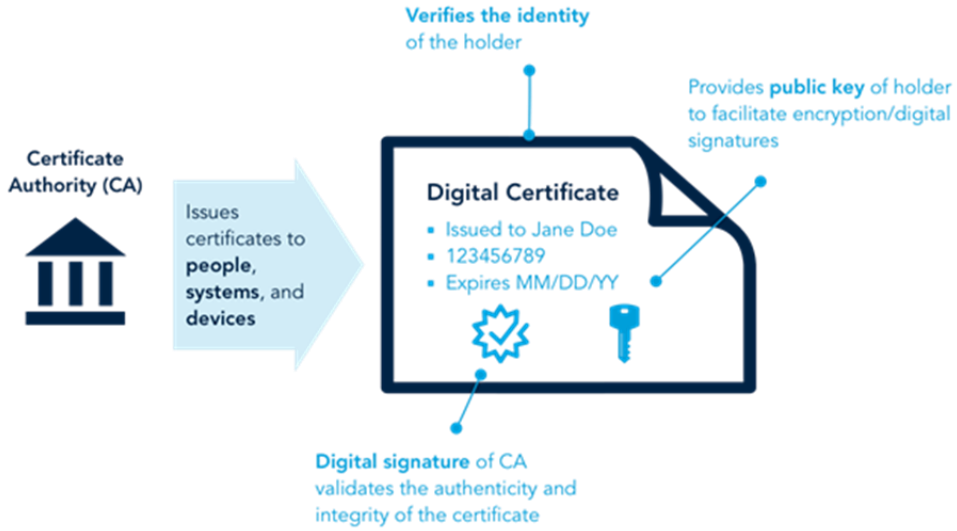
- **Authentication**: This is the process of verifying a user or system
- **Authorization**: This is the process of allowing access to protected resources



- **Encryption**: This is the process of encoding and subsequently decoding information to protect it from unauthorized individuals
- **Hashing algorithms**: These provide a way of producing a unique value for a document, and they are used in support of other security techniques

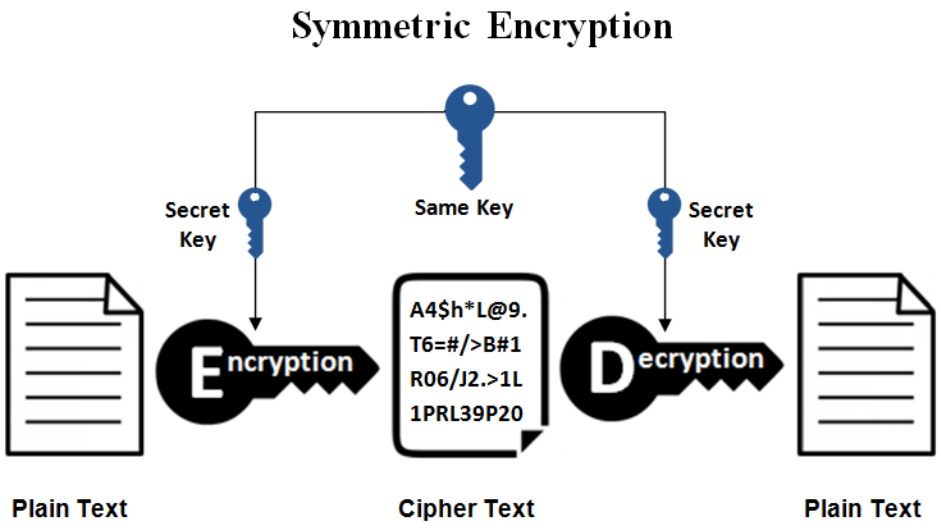


- **Digital signatures**: These provide a way of digitally authenticating a document.
- **Certificates**: These are normally used as a chain, and they support the confirmation of the identity of principals and other actors.

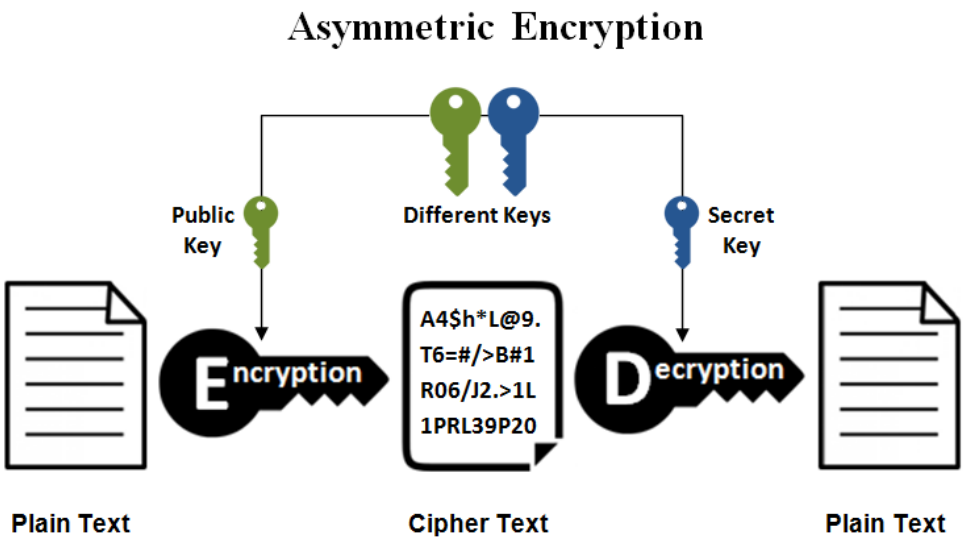


Encryption Basics

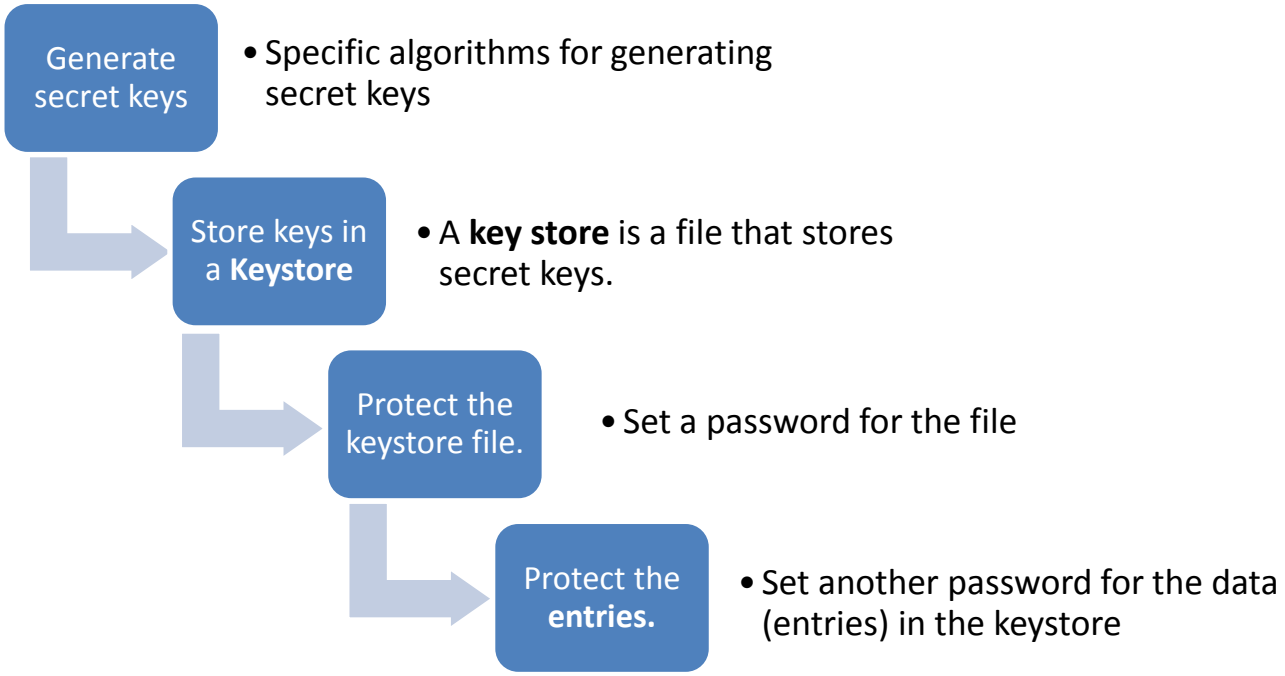
- There are two encryption techniques:
    1. **Symmetric encryption** uses a single key for encrypting and decrypting data.



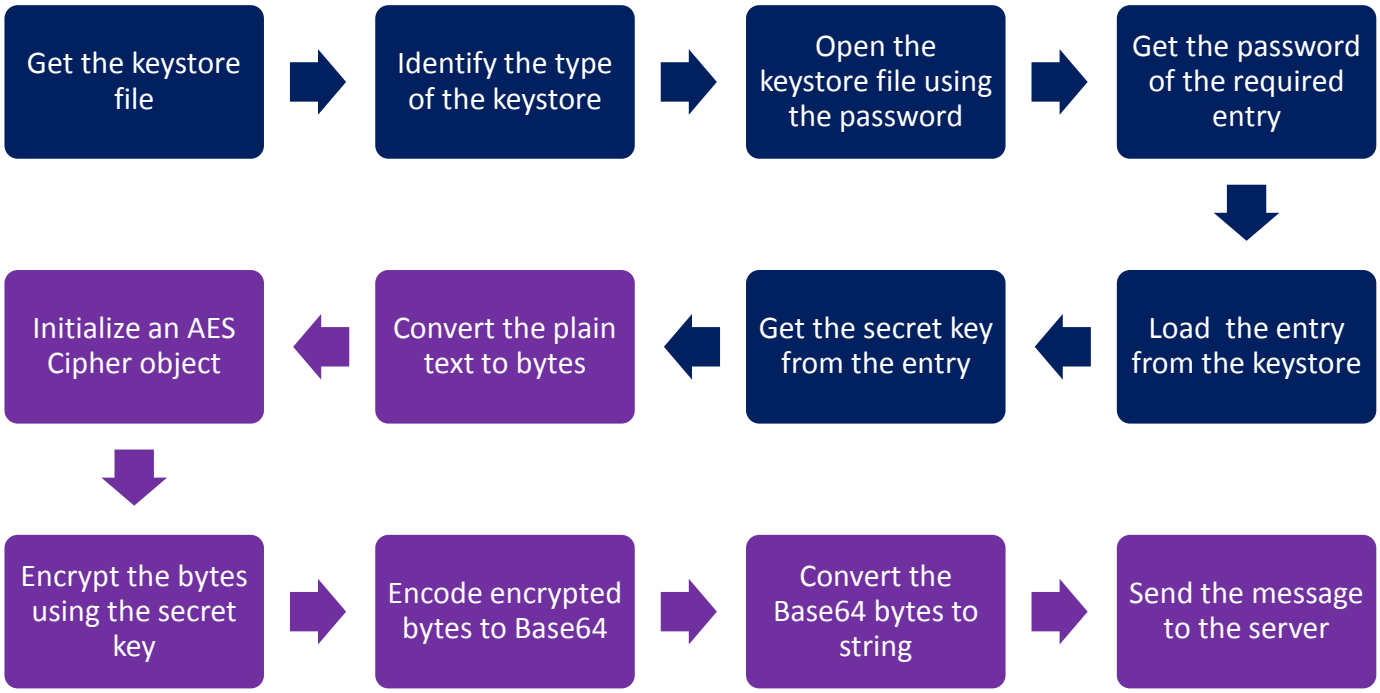
2. **Asymmetric encryption (public-key cryptography)** uses two keys; one for encrypting data called private key, and another key for decryption called public key.



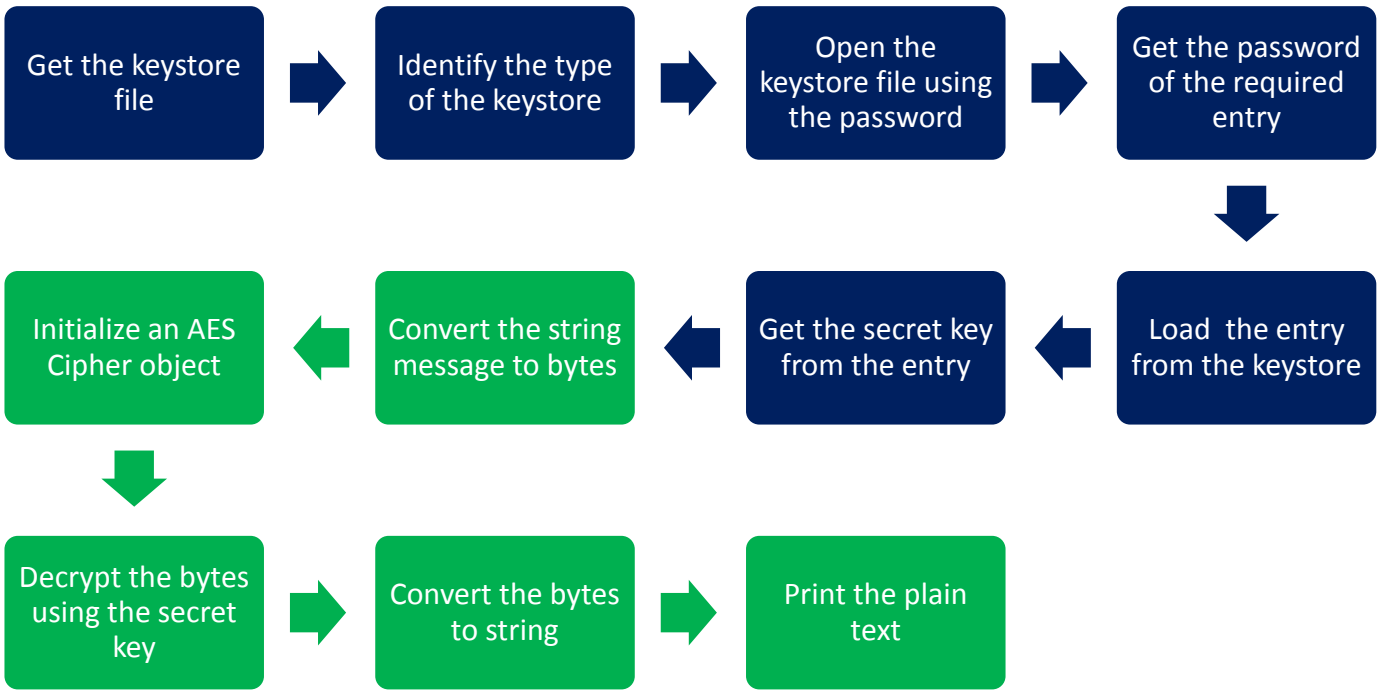
- Steps to apply encryption/decryption in a secure communication.



- Steps for an application to encrypt data.



- Steps for an application to decrypt data



Creating Keystore

- This class represents a storage facility for cryptographic keys and certificates.
- To simplify and organize the implementation details, we will create an *enum* to store several constants used for creating and accessing the keystore.

```
public enum SecureData {
    //Password for the keystore
    KEY_STORE_PASSWORD("Open ya semsem"),
    //Entry password
    KEY_PASSWORD("1234"),
    //Alias (name) of the name
    KEY_ENTRY("secret key"),
    // The name and path of the keystore file
    KEY_STORE_FILE("keystore.jks"),
    // Type of the keystore
    KEY_STORE_TYPE("JCEKS");

    public String value;

    SecureData(String value) {
        this.value = value;
    }
}
```

- Next, we will implement a class, $SymmetricKeyStore$, to create and load the key store.
- Inside the class, define the main method and another method that will create a keystore file and return it.

```
public class CreateKeyStore {
    public static KeyStore createKeyStore(String filename,
String password){

    }

    public static void main(String[] args) {

    }
}
```

- The $createKeyStore$ method takes two string parameters, the keystore filename and the password of the keystore.
- Inside the method, create a $try - catch$ block.
- Initialize a file object with the filename.
- Create a keystore object and pass the type of the keystore.

```
public static KeyStore createKeyStore(String filename, String
password){
    try{
        File file = new File(filename);
        KeyStore keyStore =
KeyStore.getInstance(SecureData.KEY_STORE_TYPE.value);

    }catch ()
}
```

- Check if the file exists, load the existing file from the hard disk using the given password. The password must be converted to a character array.
- Otherwise, store the keystore from the hard disk with the given password. The password must be converted to a character array.
- Return the keystore object.
- Handle the exceptions.

```
if(file.exists()){
        keyStore.load(new FileInputStream(file),
password.toCharArray());
    }else{
        keyStore.load(null, null);
        keyStore.store(new FileOutputStream(file),
password.toCharArray());
    }
    return keyStore;
}catch (KeyStoreException | IOException | NoSuchAlgorithmException |
CertificateException ex){
    System.out.println("Error");
}
return null;
```

- In the $main$ method, setup a $try - catch$ block.
- Create a new keystore object using the defined method.

```
public static void main(String[] args) {
    try{
        KeyStore keyStore =
createKeyStore(SecureData.KEY_STORE_FILE.value,
SecureData.KEY_STORE_PASSWORD.value);
    }catch (){

    }
}
```

- Create a new $KeyGenerator$ object for AES encryption.
- Generate a new $SecretKey$ object.

```
KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
SecretKey secretKey = keyGenerator.generateKey();
```

- Create a secret key entry in the keystore.
    - Pass the secret key object.
- Create a password for protecting that entry.
    - Pass the password for the entry in character array format.

```
KeyStore.SecretKeyEntry secretKeyEntry = new
KeyStore.SecretKeyEntry(secretKey);
KeyStore.PasswordProtection keyPassword = new

KeyStore.PasswordProtection(SecureData.KEY_PASSWORD.value.toCharArray());
```

- Write the secret key entry into the keystore using an alias name and the password.
- Store the keystore to the disk. Pass the password as a character array.

```
keyStore.setEntry(SecureData.KEY_ENTRY.value, secretKeyEntry,
passwordProtection);
keyStore.store(new FileOutputStream(SecureData.KEY_STORE_FILE.value),
        SecureData.KEY_STORE_PASSWORD.value.toCharArray());
```

- Handle the exceptions.

```
}catch (Exception ex){
    System.out.println(ex);
}
```

For the client and server, we will use the same code used in the previous labs. In addition, we will update both the client and server with the following:

1. For the client and the server, we will implement a function for reading and accessing the secret key from the keystore.
2. For the client, we will implement the $encrypt$ method for encrypting data before sending to the client.
3. For the server, we will implement the $decrypt$ method for decrypting data incoming from the client.

Accessing Keystore

- In both the client and server, create the following method for getting a secret key from the key store.
- Inside the method, initialize a $SecretKey$ object to null.
- Setup a $try - catch$ block.

```
public static SecretKey getSecretKey() {
    SecretKey secretKey = null;
    try {

    }catch (){

    }
}
```

- Inside try, create a new $File$ object pointing to the keystore file.
- Next, instantiate a $KeyStore$ object with the specified type.

```
File file = new File(SecureData.KEY_STORE_FILE.value);
KeyStore keyStore =
KeyStore.getInstance(SecureData.KEY_STORE_TYPE.value);
```

- Load the keystore file, passing its password as a character array.
- Create a $PasswordProtection$ object to access the secret key entry stored in the keystore.

```
keyStore.load(new FileInputStream(file),
SecureData.KEY_STORE_PASSWORD.value.toCharArray());
KeyStore.PasswordProtection keyPassword = new

KeyStore.PasswordProtection(SecureData.KEY_PASSWORD.value.toCharArra
y());
```

- Define an $Entry$ object to get the secret key entry from the keystore. Pass the alias of the entry and its password.
- From the entry, extract the secret key. Cast the object to $KeyStore.SecretKeyEntry$ data type.

```
KeyStore.Entry entry = keyStore.getEntry(SecureData.KEY_ENTRY.value,
passwordProtection);
secretKey = ((KeyStore.SecretKeyEntry) entry).getSecretKey();
```

- Catch the exceptions and return the secret key.

```
}catch (Exception ex){
    System.out.println("Error");
}
return secretKey;
```

Implementing Encryption at the Client

- The client application will remain the same except we add the *encrypt* method as follows.

```
public static String encrypt(String plainText, SecretKey secretKey){
    try{
        //Instantiate Cipher object of type AES
        Cipher cipher = Cipher.getInstance("AES");
        //Convert the plaintext string to bytes
        byte[] plainTextBytes = plainText.getBytes();
        //Initialize the cipher with encryption mode and the secret key
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
        //Encrypt the bytes of the plain text, returning the encrypted bytes
        byte[] encryptedBytes = cipher.doFinal(plainTextBytes);
        //Defien a Base64 encoder object
        Base64.Encoder encoder = Base64.getEncoder();
        //Encode the encrypted bytes into string
        String encryptedText = encoder.encodeToString(encryptedBytes);
        return encryptedText;
    }catch (Exception ex){
        System.out.println(ex);
    }
    return null;
}
```

- Add the highlighted modifications to the *accessServer* method of the client.

```
private static void accessServer() {
    Socket link = null;
    SecretKey secretKey = getSecretKey();
    try{
        link = new Socket(host, PORT);
        Scanner input = new Scanner(link.getInputStream());
        PrintWriter output = new PrintWriter(link.getOutputStream(), true);

        Scanner userInput = new Scanner(System.in);
        String msg, respns;

        do {
            System.out.print("Enter a message: ");
            msg = userInput.nextLine();
            String encryptedMsg = encrypt(msg, secretKey);
            output.println(encryptedMsg);
            respns = input.nextLine();
            System.out.println("\nSERVER> "+respns);
        }while (!msg.equals("***CLOSE***"));
    } catch (IOException e) {
        e.printStackTrace();
    }finally {
        try {
            System.out.println("CLOSING CONNECTION ...");
            link.close();
        } catch (IOException e) {
            System.out.println("Unable to disconnect");
            System.exit(1);
        }
    }
}
```

Implementing Decryption at Server

- The server application will remain the same except we add the *decrypt* method as follows.

```
public static String decrypt(String encryptedText, SecretKey secretKey) {
    try {
        //Instantiate Cipher object of type AES
        Cipher cipher = Cipher.getInstance("AES");
        //Define a Base64 decoder
        Base64.Decoder decoder = Base64.getDecoder();
        //Decode the encrypted message into bytes.
        byte[] encryptedBytes = decoder.decode(encryptedText);
        //Initialize the cipher object with decrypt mode and secret key
        cipher.init(Cipher.DECRYPT_MODE, secretKey);
        //Decrypt the bytes
        byte[] decryptedBytes = cipher.doFinal(encryptedBytes);
        //Convert the bytes to string
        String decryptedText = new String(decryptedBytes);
        return decryptedText;
    } catch (NoSuchAlgorithmException | NoSuchPaddingException |
             InvalidKeyException | IllegalBlockSizeException |
BadPaddingException ex) {
        System.out.println(ex);
    }
    return null;
}
```

- Add the highlighted modifications to the *handleConnection*method of the Server.

```
private static void handleConnection() {
    Socket link = null;
    SecretKey secretKey = getSecretKey();
    try {
        link = serverSocket.accept();
        Scanner input = new Scanner(link.getInputStream());
        PrintWriter output = new PrintWriter(link.getOutputStream(), true);

        int num_msgs = 0;
        String msgSecure = input.nextLine();
        String msg = decrypt(msgSecure, secretKey);
        while (!msg.equals("***CLOSE***")) {
            System.out.println("Message Received");
            System.out.println("Encrypted message: " + msgSecure);
            num_msgs++;
            output.println("Message " + num_msgs + ": " + msg);
            msgSecure = input.nextLine();
            msg = decrypt(msgSecure, secretKey);
        }
        output.println(num_msgs + " messages received");
    } catch (IOException ioex) {
        ioex.printStackTrace();
    } finally {
        try {
            System.out.println("\n*Closing Connection...*\n");
            link.close();
        } catch (IOException ioex) {
            System.out.println("Unable to disconnect!");
            System.exit(1);
        }
    }
```

CreateKeyStore class

```
public class CreateKeyStore {
    public static KeyStore createKeyStore(String filename, String password){
        try{
            File file = new File(filename);
            KeyStore keyStore =
KeyStore.getInstance(SecureData.KEY_STORE_TYPE.value);
            if(file.exists()){
                keyStore.load(new FileInputStream(file),
password.toCharArray());
            }else{
                keyStore.load(null, null);
                keyStore.store(new FileOutputStream(file),
password.toCharArray());
            }
            return keyStore;
        }catch (Exception ex){
            ex.printStackTrace();
        }
        return null;
    }

    public static void main(String[] args) {
        try{
            KeyStore keyStore =
createKeyStore(SecureData.KEY_STORE_FILE.value,
SecureData.KEY_STORE_PASSWORD.value);
            KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
            SecretKey secretKey = keyGenerator.generateKey();
            KeyStore.SecretKeyEntry secretKeyEntry = new
KeyStore.SecretKeyEntry(secretKey);
            KeyStore.PasswordProtection passwordProtection = new

KeyStore.PasswordProtection(SecureData.KEY_PASSWORD.value.toCharArray());
            keyStore.setEntry(SecureData.KEY_ENTRY.value, secretKeyEntry,
passwordProtection);
            keyStore.store(new
FileOutputStream(SecureData.KEY_STORE_FILE.value),
                    SecureData.KEY_STORE_PASSWORD.value.toCharArray());
            System.out.println("**********DONE**********");
        }catch (Exception ex){
            ex.printStackTrace();
        }
    }
}
```

TCPEchoServer

```
public class TCPEchoServer {
    private static ServerSocket serverSocket;
    private static final int PORT = 1234;

    public static String decrypt(String encryptedText, SecretKey secretKey)
{
        try {
            Cipher cipher = Cipher.getInstance("AES");
            Base64.Decoder decoder = Base64.getDecoder();
            byte[] EncryptedBytes = decoder.decode(encryptedText);
            cipher.init(Cipher.DECRYPT_MODE, secretKey);
            byte[] decryptedBytes = cipher.doFinal(EncryptedBytes);
            String msg = new String(decryptedBytes);
            return msg;
        }catch (Exception ex){
            ex.printStackTrace();
        }
        return null;
    }

    public static SecretKey getSecretKey() {
        SecretKey secretKey = null;
        try{
            File file = new File(SecureData.KEY_STORE_FILE.value);
            KeyStore keyStore =
KeyStore.getInstance(SecureData.KEY_STORE_TYPE.value);
            keyStore.load(new FileInputStream(file),
SecureData.KEY_STORE_PASSWORD.value.toCharArray());
            KeyStore.PasswordProtection passwordProtection = new

KeyStore.PasswordProtection(SecureData.KEY_PASSWORD.value.toCharArray());
            KeyStore.Entry entry =
keyStore.getEntry(SecureData.KEY_ENTRY.value, passwordProtection);
            secretKey = ((KeyStore.SecretKeyEntry)entry).getSecretKey();

        }catch (Exception ex){
            ex.printStackTrace();
        }
        return secretKey;
    }
```

Continued in the next page…

```
public static void main(String[] args) {
        System.out.println("Opening port ...");
        try {
            serverSocket = new ServerSocket(PORT);
            System.out.println("Port opened successfully...");
        } catch (IOException ioex) {
            System.out.println("Failed to connect to port: " + PORT);
            System.out.println("Please try another port");
            System.exit(1);
        }

        do {
            handleConnection();
        } while (true);
    }

    private static void handleConnection() {
        Socket link = null;
        SecretKey secretKey = getSecretKey();
        try {
            link = serverSocket.accept();
            Scanner input = new Scanner(link.getInputStream());
            PrintWriter output = new PrintWriter(link.getOutputStream(),
true);

            int num_msgs = 0;
            String msgSecure = input.nextLine();
            String msg = decrypt(msgSecure, secretKey);
            while (!msg.equals("***CLOSE***")) {
                System.out.println("Message Received");
                System.out.println("Encrypted message: " + msgSecure);
                num_msgs++;
                output.println("Message " + num_msgs + ": " + msg);
                msgSecure = input.nextLine();
                msg = decrypt(msgSecure, secretKey);
            }
            output.println(num_msgs + " messages received");
        } catch (IOException ioex) {
            ioex.printStackTrace();
        } finally {
            try {
                System.out.println("\n*Closing Connection...*\n");
                link.close();
            } catch (IOException ioex) {
                System.out.println("Unable to disconnect!");
                System.exit(1);
            }
        }
    }
}
```

TCPEchoClient

```
public class TCPEchoClient {
    private static InetAddress host;
    private static final int PORT = 1234;

    public static String encrypt(String plainText, SecretKey secretKey) {
        try {
            Cipher cipher = Cipher.getInstance("AES");
            byte[] plainTextBytes = plainText.getBytes();
            cipher.init(Cipher.ENCRYPT_MODE, secretKey);
            byte[] EncryptedBytes = cipher.doFinal(plainTextBytes);
            Base64.Encoder encoder = Base64.getEncoder();
            String encryptedText = encoder.encodeToString(EncryptedBytes);
            return encryptedText;
        }catch (Exception ex){
            ex.printStackTrace();
        }
        return null;
    }

    public static SecretKey getSecretKey() {
        SecretKey secretKey = null;
        try {
            File file = new File(SecureData.KEY_STORE_FILE.value);
            KeyStore keyStore =
KeyStore.getInstance(SecureData.KEY_STORE_TYPE.value);
            keyStore.load(new FileInputStream(file),
SecureData.KEY_STORE_PASSWORD.value.toCharArray());
            KeyStore.PasswordProtection passwordProtection =
                    new
KeyStore.PasswordProtection(SecureData.KEY_PASSWORD.value.toCharArray());
            KeyStore.Entry entry =
keyStore.getEntry(SecureData.KEY_ENTRY.value, passwordProtection);
            secretKey = ((KeyStore.SecretKeyEntry) entry).getSecretKey();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        return secretKey;
    }
```

Continued in the next page…

```
public static void main(String[] args) {
        try {
            host = InetAddress.getLocalHost();
        } catch (IOException ioex) {
            System.out.println("Host ID not found!");
            System.exit(1);
        }
        accessServer();
    }

    private static void accessServer() {
        Socket link = null;
        SecretKey secretKey = getSecretKey();
        try {
            link = new Socket(host, PORT);
            Scanner input = new Scanner(link.getInputStream());
            PrintWriter output = new PrintWriter(link.getOutputStream(),
true);

            Scanner userInput = new Scanner(System.in);
            String msg, respns;

            do {
                System.out.print("Enter a message: ");
                msg = userInput.nextLine();
                String encryptedMsg = encrypt(msg, secretKey);
                output.println(encryptedMsg);
                respns = input.nextLine();
                System.out.println("\nSERVER> " + respns);
            } while (!msg.equals("***CLOSE***"));
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                System.out.println("CLOSING CONNECTION ...");
                link.close();
            } catch (IOException e) {
                System.out.println("Unable to disconnect");
                System.exit(1);
            }
        }
    }
}
```