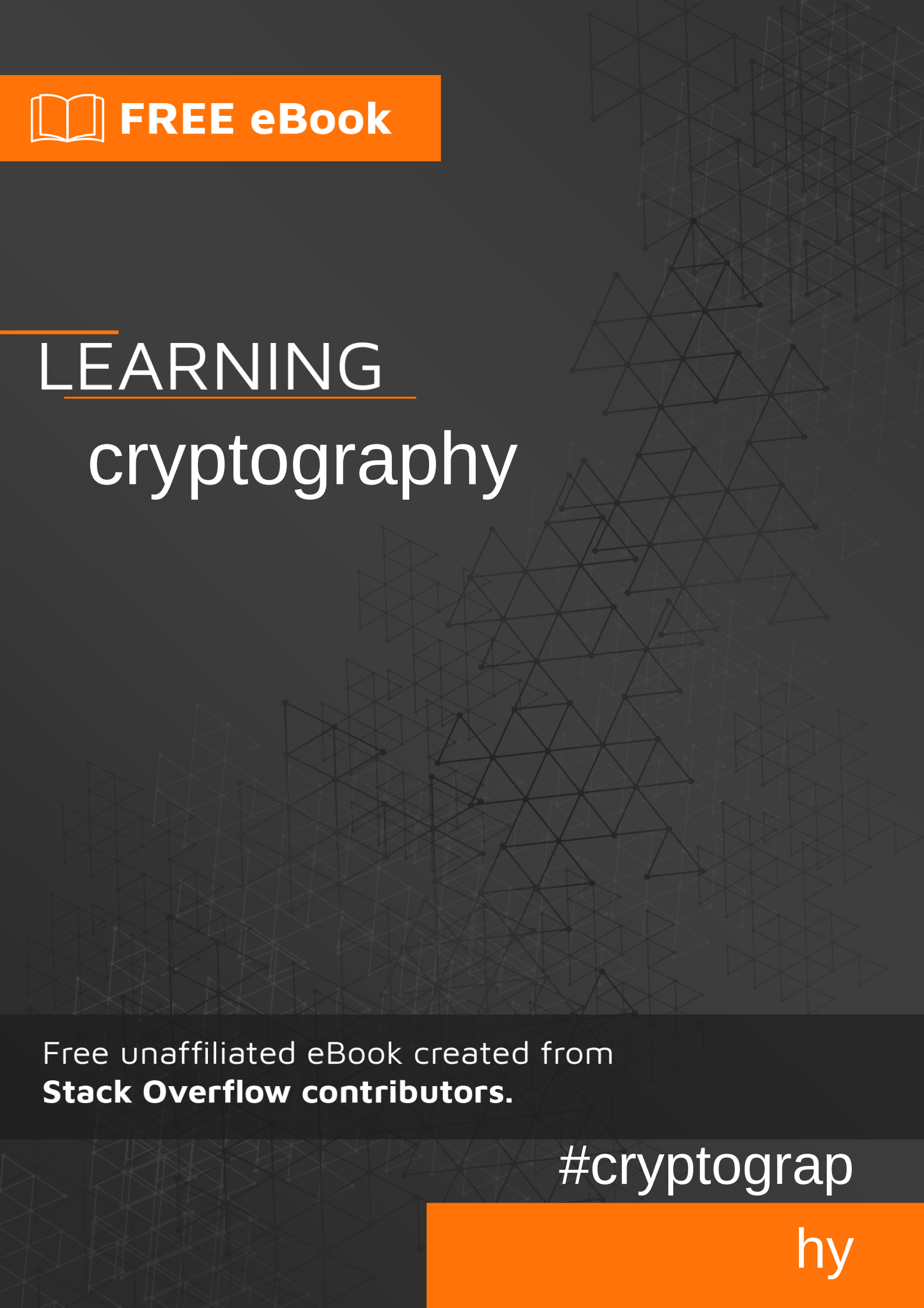

FREE eBook
LEARNING
cryptography
Free unaffiliated eBook created from
Stack Overflow contributors.
#cryptograp
hy

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: cryptography

It is an unofficial and free cryptography ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official cryptography.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with cryptography

Remarks

Modern cryptography is the cornerstone of computer and communications security. Its foundation is based on concepts of mathematics such as number theory, computational-complexity theory, and probability theory.

Cryptography deals with the securing of digital data. It refers to the design of mechanisms based on mathematical algorithms. The primary objective of using cryptography is to provide the four fundamental information security services; confidentiality, non-repudiation, authentication and data-integrity.

Examples

Integrity Validated - Symmetric Key - Encryption and Decryption example using Java

Encryption is used to transform data in its orignal format (Eg: The contents of a letter, Credentials part of authorizing a financial transaction) to something that cannot be easily reconstructed by anyone who is not intended to be part of the conversation.

Basically encryption is used to prevent eavesdropping between any two entities (individuals or a group).

In case of symmetric encryption, both the sender and receiver (Eg: Alice, Bob) must use the same encryption algorithm (generally a standardised one) and the same encryption key (known only to the two of them).

http://docs.oracle.com/javase/1.5.0/docs/guide/security/jce/JCERefGuide.html#Examples

Related Links

- https://en.wikipedia.org/wiki/History_of_cryptography
- https://en.wikipedia.org/wiki/Cryptography

```
package com.example.so.documentation.cryptography;

import java.nio.charset.Charset;
import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.spec.AlgorithmParameterSpec;
import java.util.StringTokenizer;

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
```

```
import javax.crypto.KeyGenerator;
import javax.crypto.Mac;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import javax.xml.bind.DatatypeConverter;


/**
 *
 * <p> Encryption is used to transform data in its orignal format (Eg: The contents of a
letter, Credentials part of authorizing a financial transaction) to something that
 * cannot be easily reconstructed by anyone who is not intended to be part of the
conversation.  </p>
 * <p> Basically encryption is used to prevent eavesdropping between any two entities
 * (individuals or a group). </p>
 *
 * <p> In case of symmetric encryption, both the sender and receiver (Eg: Alice, Bob) must use
the same encryption algorithm (generally a standardised one)
 * and the same encryption key (known only to the two of them). </p>
 *
 * <p> http://docs.oracle.com/javase/1.5.0/docs/guide/security/jce/JCERefGuide.html#Examples
</p>
 *
 * <p> Related Links </p>
 * <ul>
 *     <li>https://en.wikipedia.org/wiki/History_of_cryptography</li>
 *     <li>https://en.wikipedia.org/wiki/Cryptography</li>
 * </ul>
 *
 * <pre>
 *         ChangeLog : 2016-09-24
 *         1. The modified encrypted text is now reflected correctly in the log and also
updated same in javadoc comment.
 * </pre>
 * @author Ravindra HV (with inputs w.r.t integrity check from
ArtjomB[http://stackoverflow.com/users/1816580/artjom-b])
 * @since (30 July 2016)
 * @version 0.3
 *
 */
public class IntegrityValidatedSymmetricCipherExample {

    /**
     * <p>https://en.wikipedia.org/wiki/Advanced_Encryption_Standard</p>
     */
    private static final String SYMMETRIC_ENCRYPTION_ALGORITHM_NAME = "AES"; // The current
standard encryption algorithm (as of writing)

    /**
     * <p>Higher the number, the better</p>
     * <p>Encryption is performed on chunks of data defined by the key size</p>
     * <p>Higher key sizes may require modification to the JDK (Unlimited Strength
Cryptography)</p>
     *
     */
    private static final int SYMMETRIC_ENCRYPTION_KEY_SIZE = 128; // lengths can be 128, 192
and 256

    /**
```

```
     * <p>
     *         A transformation defines in what manner the encryption should be performed.
     * </p>
     * <p>
            Eg: Whether there is any link between two chunks of encrypted data (CBC) or what
should happen
     *         if there is a mismatch between the key-size and the data length.       *
     * </p>
     *
     * <p> https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation </p>
     */
    private static final String SYMMETRIC_ENCRYPTION_TRANSFORMATION = "AES/CBC/PKCS5Padding";
    private static final Charset CHARSET_INSTANCE_UTF8 = Charset.forName("UTF-8");


    private static final int AES_IV_KEY_SIZE = 128; // for AES, iv key size is fixed at 128
independent of key-size

    private static final String MAC_ALGORITHM_NAME__HMAC_SHA256 = "HmacSHA256";
    private static final String HASH_FIELDS_SEPARATOR = "|" ;




    /**
     * @param args
     * <p>Sample output.</p>
     * <pre>
Encrypted, Base64 encoded text
:W1DePjeYMlI6xmyq9jr+cw==|55F80F4C2987CC143C69563025FACE22|GLR3T8GdcocpsTM1qSXp5jLsNx6QRK880BtgnV1jFg0=

Decrypted text :helloworld
Encrypted, Base64 encoded text -
v2:1XX/A9BO1Cp8mK+SHh9iHA==|B8294AC9967BB57D714ACCB3EE5710BD|TnjdaWbvp+H6yCbAAQFMkWNixeW8VwmW48YlKA/AAy

Decrypted text  - v2:helloworld
Encrypted, Base64 encoded text - v3
(original):EU4+rAZ2vOKtoSDiDPcO+A==|AEEB8DD341D8D9CD2EDFA05A4595EBD2|7anESSSJf1dHobS5tDdQ1mCNkFcIgCvtNC

Encrypted, Base64 encoded text - v3
(modified):FU4+rAZ2vOKtoSDiDPcO+A==|AEEB8DD341D8D9CD2EDFA05A4595EBD2|7anESSSJf1dHobS5tDdQ1mCNkFcIgCvtNC

Error : Integrity check failed
Exception in thread "main" java.lang.RuntimeException: Error : Integrity check failed
    at
com.example.so.documentation.cryptography.IntegrityValidatedSymmetricCipherExampleThree.decrypt(Integri

    at
com.example.so.documentation.cryptography.IntegrityValidatedSymmetricCipherExampleThree.main(IntegrityV

     * </pre>
     */
    public static void main(String[] args) {

        /*
         * EncryptionKey : Shared secret between receiver and sender (who generates the
password and how its shared depends on the purpose)
         * This program generates a new one every time its run !
         * Normally it would be generated once and then be stored somewhere (Eg: In a JCEKS
keystore file).
         */
```

```
        byte[] generatedSharedSecret = secretKeyGeneratorUtility();
        byte[] generatedSharedHMACKey = secretKeyGeneratorUtility();
        String plainText = "helloworld";

        String encryptedText = encrypt(plainText, generatedSharedSecret,
generatedSharedHMACKey);
        System.out.println("Encrypted, Base64 encoded text :"+encryptedText);
        String decryptedText = decrypt(encryptedText, generatedSharedSecret,
generatedSharedHMACKey);
        System.out.println("Decrypted text :"+decryptedText);

        String encryptedTextTwo = encrypt(plainText, generatedSharedSecret,
generatedSharedHMACKey);
        System.out.println("Encrypted, Base64 encoded text - v2:"+encryptedTextTwo);
        String decryptedTextTwo = decrypt(encryptedTextTwo, generatedSharedSecret,
generatedSharedHMACKey);
        System.out.println("Decrypted text  - v2:"+decryptedTextTwo);

        String encryptedTextThree = encrypt(plainText, generatedSharedSecret,
generatedSharedHMACKey);
        System.out.println("Encrypted, Base64 encoded text - v3
(original):"+encryptedTextThree);
        char[] encryptedTextThreeChars = encryptedTextThree.toCharArray();
        encryptedTextThreeChars[0] = (char) ((encryptedTextThreeChars[0])+1);
        String encryptedTextThreeModified = new String(encryptedTextThreeChars);
        System.out.println("Encrypted, Base64 encoded text - v3
(modified):"+encryptedTextThreeModified);

        String decryptedTextThree = decrypt(encryptedTextThreeModified, generatedSharedSecret,
generatedSharedHMACKey);
        System.out.println("Decrypted text  - v3:"+decryptedTextThree);

    }


    public static String encrypt(String plainText, byte[] key, byte[] hmacKey) {

        byte[] plainDataBytes = plainText.getBytes(CHARSET_INSTANCE_UTF8);
        byte[] iv = initializationVectorGeneratorUtility();
        byte[] encryptedDataBytes = encrypt(plainDataBytes, key, iv);

        String initializationVectorHex = DatatypeConverter.printHexBinary(iv);
        String encryptedBase64EncodedString =
DatatypeConverter.printBase64Binary(encryptedDataBytes); // Generally the encrypted data is
encoded in Base64 or hexadecimal encoding for ease of handling.
        String hashInputString = encryptedBase64EncodedString + HASH_FIELDS_SEPARATOR +
initializationVectorHex + HASH_FIELDS_SEPARATOR;
        String hashedOutputString =
DatatypeConverter.printBase64Binary(messageHashWithKey(hmacKey,
hashInputString.getBytes(CHARSET_INSTANCE_UTF8)));
        String encryptionResult = hashInputString + hashedOutputString;
        return encryptionResult;
    }

    public static byte[] encrypt(byte[] plainDataBytes, byte[] key, byte[] iv) {
        byte[] encryptedDataBytes = encryptOrDecrypt(plainDataBytes, key, iv, true);
        return encryptedDataBytes;
    }


    public static String decrypt(String cipherInput, byte[] key, byte[] hmacKey) {
```

```
        StringTokenizer stringTokenizer = new StringTokenizer(cipherInput,
HASH_FIELDS_SEPARATOR);

        String encryptedString = stringTokenizer.nextToken();
        String initializationVectorHex = stringTokenizer.nextToken();
        String hashedString = stringTokenizer.nextToken();

        String hashInputString = encryptedString + HASH_FIELDS_SEPARATOR +
initializationVectorHex + HASH_FIELDS_SEPARATOR;
        String hashedOutputString =
DatatypeConverter.printBase64Binary(messageHashWithKey(hmacKey,
hashInputString.getBytes(CHARSET_INSTANCE_UTF8)));

        if( hashedString.equals(hashedOutputString) == false ) {
            String message = "Error : Integrity check failed";
            System.out.println(message);
            throw new RuntimeException(message);
        }

        byte[] encryptedDataBytes = DatatypeConverter.parseBase64Binary(encryptedString); //
The Base64 encoding must be reversed so as to reconstruct the raw bytes.
        byte[] iv = DatatypeConverter.parseHexBinary(initializationVectorHex);
        byte[] plainDataBytes = decrypt(encryptedDataBytes, key, iv);
        String plainText = new String(plainDataBytes, CHARSET_INSTANCE_UTF8);
        return plainText;
    }

    public static byte[] decrypt(byte[] encryptedDataBytes, byte[] key, byte[] iv) {
        byte[] decryptedDataBytes = encryptOrDecrypt(encryptedDataBytes, key, iv, false);
        return decryptedDataBytes;
    }


    public static byte[] encryptOrDecrypt(byte[] inputDataBytes, byte[] key, byte[] iv,
boolean encrypt) {
        byte[] resultDataBytes = null;

        // Exceptions, if any, are just logged to console for this example.
        try {
            Cipher cipher = Cipher.getInstance(SYMMETRIC_ENCRYPTION_TRANSFORMATION);
            SecretKey secretKey = new SecretKeySpec(key, SYMMETRIC_ENCRYPTION_ALGORITHM_NAME);
            AlgorithmParameterSpec algorithmParameterSpec = new IvParameterSpec(iv);
            if(encrypt) {
                cipher.init(Cipher.ENCRYPT_MODE, secretKey, algorithmParameterSpec);
            }
            else {
                cipher.init(Cipher.DECRYPT_MODE, secretKey, algorithmParameterSpec);
            }

            resultDataBytes = cipher.doFinal(inputDataBytes); // In relative terms, invoking
do-final in one go is fine as long as the input size is small.
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        } catch (NoSuchPaddingException e) {
            e.printStackTrace();
        } catch (InvalidKeyException e) {
            e.printStackTrace();
        } catch (IllegalBlockSizeException e) {
            e.printStackTrace();
        } catch (BadPaddingException e) {
            e.printStackTrace();
```

```
        } catch (InvalidAlgorithmParameterException e) {
            e.printStackTrace();
        }

        return resultDataBytes;
    }


    private static byte[] secretKeyGeneratorUtility() {
        byte[] keyBytes = null;

        try {
            KeyGenerator keyGenerator =
KeyGenerator.getInstance(SYMMETRIC_ENCRYPTION_ALGORITHM_NAME);
            keyGenerator.init(SYMMETRIC_ENCRYPTION_KEY_SIZE);
            SecretKey secretKey = keyGenerator.generateKey();
            keyBytes = secretKey.getEncoded();
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }

        return keyBytes;

    }


    /**
     * <p> InitialVector : Helps in avoiding generating the same encrypted result, even when
the same encryption - algorithm and key are used. </p>
     * <p> Since this is also required to be known to both sender and receiver, its either
based on some convention or is part of the cipher-text transmitted.</p>
     * <p> https://en.wikipedia.org/wiki/Initialization_vector </p>
     * @return
     */
    private static byte[] initializationVectorGeneratorUtility() {
        byte[] initialVectorResult = null;

        try {
            KeyGenerator keyGenerator =
KeyGenerator.getInstance(SYMMETRIC_ENCRYPTION_ALGORITHM_NAME);
            keyGenerator.init(AES_IV_KEY_SIZE);
            SecretKey secretKey = keyGenerator.generateKey();
            initialVectorResult = secretKey.getEncoded();
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }

        return initialVectorResult;
    }


    private static byte[] messageHashWithKey(byte[] key, byte[] data) { // byte[] iv,
        byte[] hmac = null;

        try {
            Mac mac = Mac.getInstance(MAC_ALGORITHM_NAME__HMAC_SHA256);
            SecretKeySpec secretKeySpec = new SecretKeySpec(key,
MAC_ALGORITHM_NAME__HMAC_SHA256);
            //AlgorithmParameterSpec algorithmParameterSpec = new IvParameterSpec(iv);
            mac.init(secretKeySpec); // algorithmParameterSpec
            hmac = mac.doFinal(data);
```

```
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        } catch (InvalidKeyException e) {
            e.printStackTrace();
        } /*catch (InvalidAlgorithmParameterException e) {
            e.printStackTrace();
        }*/

        return hmac;

    }

}
```

Introduction

Cryptography is the science of using mathematical constructs (codes) to make communication secure. The field of cryptography is a subset of the field of Information Security.

There are many cryptographic operations possible; some best known examples are:

- **Encryption :** transforming a plaintext message into a ciphertext message so that the the message remains *confidential*
- **Decryption :** transforming a ciphertext message back into a plaintext message
- **Secure hashing :** performing irreversible (one-way) compression to create a statically sized, computationally distinct representation for a specific message.

Cryptography is based on math, and arithmetic is frequently used in algorithms related to cryptography. There is a small subset of primitives, schemes and protocols that are used by developers. Developers usually do not implement the algorithms themselves but use the schemes and protocols provided by cryptographic API's and runtimes.

A **primitive** could be a block cipher such as AES. A primitive is any algorithm that is used as building block for a cryptographic scheme. A **scheme** is for instance a block cipher *mode of operation* such as CBC or GCM. One or more cryptographic schemes can make up a cryptographic protocol. A **protocol** such as TLS uses many cryptographic schemes, but also message encoding / decoding techniques, message ordering, conditions for use etc. Low level cryptographic API's just provide direct access to primitives while high level API's may offer access to full protocol implementations.

Messages have been encrypted and decrypted by hand since written word was invented. Mechanical devices have been used at least since ancient Greek society. This kind of cryptography is referred to as *classic cryptography*. Many introductions to cryptography start with classic cryptography as it is relatively easy to analyze. Classic algorithms however do not adhere to the security required from modern constructs and are often easy to break. Examples of classic schemes are the Caesar and Vigenère. The most well known mechanical device is undoubtedly the Enigma coding machine.

Modern cryptography is based on science - mainly math and number/group theory. It involves much more intricate algorithms and key sizes. These can only be handled efficiently by computing devices. For this reason modern cryptography mainly uses byte oriented input and output. This

means that messages need to be converted to binary and back before they can be transformed by any implementation of a cryptographic algorithm. This means that (textual) messages need to be transformed using character-encoding before being encrypted.

Forms of *character-encoding* of textual messages is UTF-8. Structured messages may be encoded using ASN.1 / DER or *canonical* XML representations - or any number of proprietary techniques. Sometimes the binary output needs to be transformed back into text as well. In this case an *encoding* scheme such as base 64 or hexadecimals can be used to represent the binary data within text.

Cryptography is notoriously hard to get right. Developers should only use constructs that they fully understand. If possible, a developer should use a higher level protocol such as TLS to create transport security. There is no practical chance of creating a secure algorithm, scheme or protocol without formal education or extensive experience. Copy/pasting examples from the internet is not likely lead to secure solutions and may even result in data loss.

Read Getting started with cryptography online:
https://riptutorial.com/cryptography/topic/3408/getting-started-with-cryptography

Chapter 2: Caesar cipher

Introduction

It is the simple shift monoalphabetic classical cipher where each letter is replaced by a letter 3 position (actual Caesar cipher) ahead using the circular alphabetic ordering i.e. letter after Z is A. So when we encode HELLO WORLD, the cipher text becomes KHOORZRUOG.

Examples

Introduction

The Caesar cipher is a classic encryption method. It works by shifting the characters by a certain amount. For example, if we choose a shift of 3, A will become D and E will become H.

The following text has been encrypted using a 23 shift.

```
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD
```

Python implementation

The ASCII way

This shifts the characters but doesn't care if the new character is not a letter. This is good if you want to use punctuation or special characters, but it won't necessarily give you letters only as an output. For example, "z" 3-shifts to "}".

```
def ceasar(text, shift):
    output = ""
    for c in text:
        output += chr(ord(c) + shift)
    return output
```

ROT13

ROT13 is a special case of Caesar cipher, with a 13 shift. Only letters are changed, and white-space and special characters are left as they are.

What is interesting is that ROT13 is a reciprocal cipher : applying ROT13 twice will give you the initial input. Indeed, 2 * 13 = 26, the number of letters in the alphabet.

As ROT13 doesn't have a key as input parameter it is often seen more as a *encoding algorithm* or, more specifically, an *obfuscation algorithm* rather than a cipher.

ROT13 just makes it hard to read messages directly and is therefore often used for offensive messages or puns of jokes. It doesn't provide any computational security.

A Java implementation for Caesar Cipher

Implementation of the Caesar cipher.

- This implementation performs the shift operation only on upper and lower case alphabets and retains the other characters (such as space as-is).
- The Caesar cipher is not secure as per current standards.
- Below example is for illustrative purposes only !
- Reference: https://en.wikipedia.org/wiki/Caesar_cipher

```
package com.example.so.cipher;

/**
 * Implementation of the Caesar cipher.
 * <p>
 * <ul>
 * <li>This implementation performs the shift operation only on upper and lower
 * case alphabets and retains the other characters (such as space as-is).</li>
 * <li>The Caesar cipher is not secure as per current standards.</li>
 * <li>Below example is for illustrative purposes only !</li>
 * <li>Reference: https://en.wikipedia.org/wiki/Caesar_cipher</li>
 * </ul>
 * </p>
 *
 * @author Ravindra HV
 * @author Maarten Bodewes (beautification only)
 * @since 2016-11-21
 * @version 0.3
 *
 */
public class CaesarCipher {

    public static final char START_LOWER_CASE_ALPHABET = 'a'; // ASCII-97
    public static final char END_LOWER_CASE_ALPHABET = 'z';   // ASCII-122

    public static final char START_UPPER_CASE_ALPHABET = 'A'; // ASCII-65
    public static final char END_UPPER_CASE_ALPHABET = 'Z';   // ASCII-90

    public static final int ALPHABET_SIZE = 'Z' - 'A' + 1;    // 26 of course

    /**
     * Performs a single encrypt followed by a single decrypt of the Caesar
     * cipher, prints out the intermediate values and finally validates
     * that the decrypted plaintext is identical to the original plaintext.
     *
     * <p>
     * This method outputs the following:
     *
     * <pre>
     * Plaintext  : The quick brown fox jumps over the lazy dog
     * Ciphertext : Qeb nrfzh yoltk clu grjmp lsbo qeb ixwv ald
     * Decrypted  : The quick brown fox jumps over the lazy dog
```

```
     * Successful decryption: true
     * </pre>
     * </p>
     *
     * @param args (ignored)
     */
    public static void main(String[] args) {

        int shift = 23;
        String plainText = "The quick brown fox jumps over the lazy dog";

        System.out.println("Plaintext  : " + plainText);

        String ciphertext = caesarCipherEncrypt(plainText, shift);
        System.out.println("Ciphertext : " + ciphertext);

        String decrypted = caesarCipherDecrypt(ciphertext, shift);
        System.out.println("Decrypted  : " + decrypted);
        System.out.println("Successful decryption: "
                + decrypted.equals(plainText));
    }

    public static String caesarCipherEncrypt(String plaintext, int shift) {
        return caesarCipher(plaintext, shift, true);
    }

    public static String caesarCipherDecrypt(String ciphertext, int shift) {
        return caesarCipher(ciphertext, shift, false);
    }

    private static String caesarCipher(
            String input, int shift, boolean encrypt) {

        // create an output buffer of the same size as the input
        StringBuilder output = new StringBuilder(input.length());

        for (int i = 0; i < input.length(); i++) {

            // get the next character
            char inputChar = input.charAt(i);

            // calculate the shift depending on whether to encrypt or decrypt
            int calculatedShift = (encrypt) ? shift : (ALPHABET_SIZE - shift);

            char startOfAlphabet;
            if ((inputChar >= START_LOWER_CASE_ALPHABET)
                    && (inputChar <= END_LOWER_CASE_ALPHABET)) {

                // process lower case
                startOfAlphabet = START_LOWER_CASE_ALPHABET;
            } else if ((inputChar >= START_UPPER_CASE_ALPHABET)
                    && (inputChar <= END_UPPER_CASE_ALPHABET)) {

                // process upper case
                startOfAlphabet = START_UPPER_CASE_ALPHABET;
            } else {

                // retain all other characters
                output.append(inputChar);

                // and continue with the next character
```

```
                continue;
            }

            // index the input character in the alphabet with 0 as base
            int inputCharIndex =
                    inputChar - startOfAlphabet;

            // cipher / decipher operation (rotation uses remainder operation)
            int outputCharIndex =
                    (inputCharIndex + calculatedShift) % ALPHABET_SIZE;

            // convert the new index in the alphabet to an output character
            char outputChar =
                    (char) (outputCharIndex + startOfAlphabet);

            // add character to temporary-storage
            output.append(outputChar);
        }

        return output.toString();
    }
}
```

Program Output:

```
Plaintext  : The quick brown fox jumps over the lazy dog
Ciphertext : Qeb nrfzh yoltk clu grjmp lsbo qeb ixwv ald
Decrypted  : The quick brown fox jumps over the lazy dog
Successful decryption: true
```

Python implementation

The following code example implements the Caesar cipher and shows the properties of the cipher.

It handles both uppercase and lowercase alpha-numerical characters, leaving all other characters as they were.

The following properties of the Caesar cipher are shown:

- weak keys;
- low key space;
- the fact that each key has a reciprocal (inverse) key;
- the relation with ROT13;

it also shows the following - more generic - cryptographic notions:

- weak keys;
- the difference between obfuscation (without a key) and encryption;
- brute forcing a key;
- the missing integrity of the ciphertext.

```
def caesarEncrypt(plaintext, shift):
```

```
    return caesarCipher(True, plaintext, shift)

def caesarDecrypt(ciphertext, shift):
    return caesarCipher(False, ciphertext, shift)

def caesarCipher(encrypt, text, shift):
    if not shift in range(0, 25):
        raise Exception('Key value out of range')

    output = ""
    for c in text:
        # only encrypt alphanumerical characters
        if c.isalpha():
            # we want to shift both upper- and lowercase characters
            ci = ord('A') if c.isupper() else ord('a')

            # if not encrypting, we're decrypting
            if encrypt:
                output += caesarEncryptCharacter(c, ci, shift)
            else:
                output += caesarDecryptCharacter(c, ci, shift)
        else:
            # leave other characters such as digits and spaces
            output += c
    return output

def caesarEncryptCharacter(plaintextCharacter, positionOfAlphabet, shift):
    # convert character to the (zero-based) index in the alphabet
    n = ord(plaintextCharacter) - positionOfAlphabet
    # perform the >positive< modular shift operation on the index
    # this always returns a value within the range [0, 25]
    # (note that 26 is the size of the western alphabet)
    x = (n + shift) % 26 # <- the magic happens here
    # convert the index back into a character
    ctc = chr(x + positionOfAlphabet)
    # return the result
    return ctc

def caesarDecryptCharacter(plaintextCharacter, positionOfAlphabet, shift):
    # convert character to the (zero-based) index in the alphabet
    n = ord(plaintextCharacter) - positionOfAlphabet
    # perform the >negative< modular shift operation on the index
    x = (n - shift) % 26
    # convert the index back into a character
    ctc = chr(x + positionOfAlphabet)
    # return the result
    return ctc

def encryptDecrypt():
    print '--- Run normal encryption / decryption'
    plaintext = 'Hello world!'
    key = 3 # the original value for the Caesar cipher
    ciphertext = caesarEncrypt(plaintext, key)
    print ciphertext
    decryptedPlaintext = caesarDecrypt(ciphertext, key)
    print decryptedPlaintext
encryptDecrypt()

print '=== Now lets show some cryptographic properties of the Caesar cipher'

def withWeakKey():
```

```
    print '--- Encrypting plaintext with a weak key is not a good idea'
    plaintext = 'Hello world!'
    # This is the weakest key of all, it does nothing
    weakKey = 0
    ciphertext = caesarEncrypt(plaintext, weakKey)
    print ciphertext # just prints out the plaintext
withWeakKey();

def withoutDecrypt():
    print '--- Do we actually need caesarDecrypt at all?'
    plaintext = 'Hello world!'
    key = 3 # the original value for the Caesar cipher
    ciphertext = caesarEncrypt(plaintext, key)
    print ciphertext
    decryptionKey = 26 - key; # reciprocal value
    decryptedPlaintext = caesarEncrypt(ciphertext, decryptionKey)
    print decryptedPlaintext # performed decryption
withoutDecrypt()


def punnify():
    print '--- ROT 13 is the Caesar cipher with a given, reciprocal, weak key: 13'
    # The key is weak because double encryption will return the plaintext
    def rot13(pun):
        return caesarEncrypt(pun, 13)

    print 'Q: How many marketing people does it take to change a light bulb?'
    obfuscated = 'N: V jvyy unir gb trg onpx gb lbh ba gung.'
    print obfuscated
    deobfuscated = rot13(obfuscated)
    print deobfuscated
    # We should not leak the pun, right? Lets obfuscate afterwards!
    obfuscatedAgain = rot13(deobfuscated)
    print obfuscatedAgain
punnify()

def bruteForceAndLength():
    print '--- Brute forcing is very easy as there are only 25 keys in the range [1..25]'
    # Note that AES-128 has 340,282,366,920,938,463,463,374,607,431,768,211,456 keys
    # and is therefore impossible to bruteforce (if the key is correctly generated)
    key = 10;
    plaintextToFind = 'Hello Maarten!'
    ciphertextToBruteForce = caesarEncrypt(plaintextToFind, key)
    for candidateKey in range(1, 25):
        bruteForcedPlaintext = caesarDecrypt(ciphertextToBruteForce, candidateKey)
        # lets assume the adversary knows 'Hello', but not the name
        if bruteForcedPlaintext.startswith('Hello'):
            print 'key value: ' + str(candidateKey) + ' gives : ' + bruteForcedPlaintext

    print '--- Length of plaintext usually not hidden'
    # Side channel attacks on ciphertext lengths are commonplace! Beware!
    if len(ciphertextToBruteForce) != len('Hello Stefan!'):
        print 'The name is not Stefan (but could be Stephan)'
bruteForceAndLength()

def manInTheMiddle():
    print '--- Ciphers are vulnerable to man-in-the-middle attacks'
    # Hint: do not directly use a cipher for transport security
    moneyTransfer = 'Give Maarten one euro'
    key = 1
    print moneyTransfer
```

```
    encryptedMoneyTransfer = caesarEncrypt(moneyTransfer, key)
    print encryptedMoneyTransfer
    # Man in the middle replaces third word with educated guess
    # (or tries different ciphertexts until success)
    encryptedMoneyTransferWords  = encryptedMoneyTransfer.split(' ');
    encryptedMoneyTransferWords[2] = 'ufo' # unidentified financial object
    modifiedEncryptedMoneyTransfer = ' '.join(encryptedMoneyTransferWords)
    print modifiedEncryptedMoneyTransfer
    decryptedMoneyTransfer = caesarDecrypt(modifiedEncryptedMoneyTransfer, key)
    print decryptedMoneyTransfer
manInTheMiddle()
```

Read Caesar cipher online: https://riptutorial.com/cryptography/topic/7504/caesar-cipher

Chapter 3: Playfair Cipher

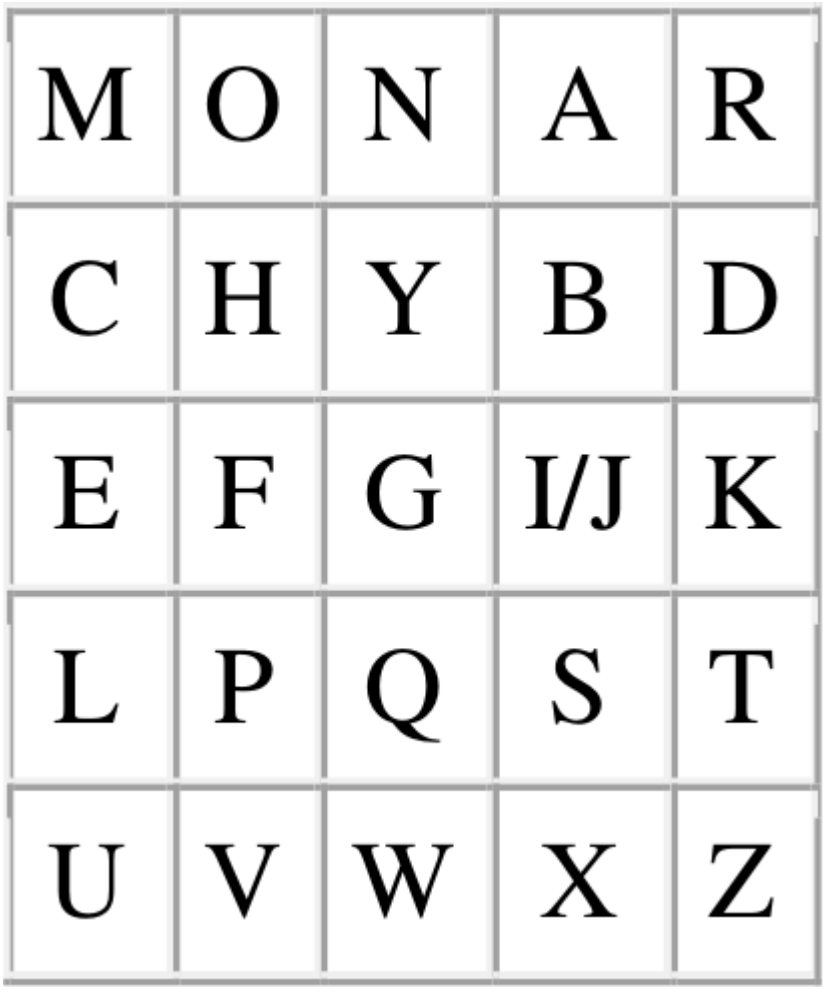
Introduction

The best-known multiple-letter encryption cipher is the Playfair, which treats digrams in the plaintext as single units and translates these units into ciphertext digrams The Playfair algorithm is based on the use of a 5 x 5 matrix of letters constructed using a keyword.

Examples

Example of Playfair Cipher Encryption along with Encryption and Decryption Rule

A keyword is "MONARCHY" then the matrix will look like



The matrix is constructed by filling in the letters of the keyword (minus duplicates) from left to right and from top to bottom, and then filling in the remainder of the matrix with the remaining letters in alphabetic order. Plaintext is encrypted two letters at a time, according to the following rules:

1. Take the characters in the text(plain/cipher) and make a group of two characters. If the numbers of characters in the text is odd then add a filler letter (usually we use 'x').

Text="HELLO" then it would be group as
HE | LL | OX

2. Two plaintext letters that fall in the same row of the matrix are each replaced by the letter to the right, with the first element of the row circularly following the last. For example, ar is encrypted as RM.

3. Two plaintext letters that fall in the same column are each replaced by the letter beneath, with the top element of the column circularly following the last. For example, mu is encrypted as CM.

4. Otherwise, each plaintext letter in a pair is replaced by the letter that lies in its own row and the column occupied by the other plaintext letter. Thus, hs becomes BP and ea becomes IM (or JM, as the encipherer wishes).

A Simple Java Program that implements the Playfair Cipher is given below:

```
import java.util.Scanner;


public class Playfair {
public static void main(String[] args) {
    Scanner in=new Scanner(System.in);

    System.out.print("Enter keyword: ");
    String key=in.nextLine();
    System.out.print("Enter message to encrypt: ");
    String msg=in.nextLine();

    PFEncryption pfEncryption=new PFEncryption();
    pfEncryption.makeArray(key);
    msg=pfEncryption.manageMessage(msg);
    pfEncryption.doPlayFair(msg, "Encrypt");
    String en=pfEncryption.getEncrypted();
    System.out.println("Encrypting....\n\nThe encrypted text is: " + en);
    System.out.println("=====================================");
    pfEncryption.doPlayFair(en, "Decrypt");
    System.out.print("\nDecrypting....\n\nThe encrypted text is: " +
pfEncryption.getDecrypted());
}
}

class PFEncryption{

private char [][] alphabets= new char[5][5];
private char[] uniqueChar= new char[26];
private String ch="ABCDEFGHIKLMNOPQRSTUVWXYZ";
private String encrypted="";
private String decrypted="";

void makeArray(String keyword){
    keyword=keyword.toUpperCase().replace("J","I");
    boolean present, terminate=false;
    int val=0;
    int uniqueLen;
    for (int i=0; i<keyword.length(); i++){
        present=false;
```

```
        uniqueLen=0;
        if (keyword.charAt(i)!= ' '){
            for (int k=0; k<uniqueChar.length; k++){
                if (Character.toString(uniqueChar[k])==null){
                    break;
                }
                uniqueLen++;
            }
            for (int j=0; j<uniqueChar.length; j++){
                if (keyword.charAt(i)==uniqueChar[j]){
                    present=true;
                }
            }
            if (!present){
                uniqueChar[val]=keyword.charAt(i);
                val++;
            }
        }
        ch=ch.replaceAll(Character.toString(keyword.charAt(i)), "");
    }

    for (int i=0; i<ch.length(); i++){
        uniqueChar[val]=ch.charAt(i);
        val++;
    }
    val=0;

    for (int i=0; i<5; i++){
        for (int j=0; j<5; j++){
            alphabets[i][j]=uniqueChar[val];
            val++;
            System.out.print(alphabets[i][j] + "\t");
        }
        System.out.println();
    }
    }

    String manageMessage(String msg){
    int val=0;
    int len=msg.length()-2;
    String newTxt="";
    String intermediate="";
    while (len>=0){
        intermediate=msg.substring(val, val+2);
        if (intermediate.charAt(0)==intermediate.charAt(1)){
            newTxt=intermediate.charAt(0) + "x" + intermediate.charAt(1);
            msg=msg.replaceFirst(intermediate, newTxt);
            len++;
        }
        len-=2;
        val+=2;
    }

    if (msg.length()%2!=0){
        msg=msg+'x';
        }
        return msg.toUpperCase().replaceAll("J","I").replaceAll(" ","");
    }

void doPlayFair(String msg, String tag){
    int val=0;
```

```
    while (val<msg.length()){
        searchAndEncryptOrDecrypt(msg.substring(val, val + 2), tag);
        val+=2;
    }
}

void searchAndEncryptOrDecrypt(String doubblyCh, String tag){
    char ch1=doubblyCh.charAt(0);
    char ch2=doubblyCh.charAt(1);
    int row1=0, col1=0, row2=0, col2=0;
    for (int i=0; i<5; i++){
        for (int j=0; j<5; j++){
            if (alphabets[i][j]==ch1){
                row1=i;
                col1=j;
            }else if (alphabets[i][j]==ch2){
                row2=i;
                col2=j;
            }
        }
    }
    if (tag=="Encrypt")
        encrypt(row1, col1, row2, col2);
    else if(tag=="Decrypt")
        decrypt(row1, col1, row2, col2);
}

void encrypt(int row1, int col1, int row2, int col2){
    if (row1==row2){
        col1=col1+1;
        col2=col2+1;
        if (col1>4)
            col1=0;
        if (col2>4)
            col2=0;

encrypted+=(Character.toString(alphabets[row1][col1])+Character.toString(alphabets[row1][col2]));

    }else if(col1==col2){
        row1=row1+1;
        row2=row2+1;
        if (row1>4)
            row1=0;
        if (row2>4)
            row2=0;

encrypted+=(Character.toString(alphabets[row1][col1])+Character.toString(alphabets[row2][col1]));

    }else{

encrypted+=(Character.toString(alphabets[row1][col2])+Character.toString(alphabets[row2][col1]));

    }
}

void decrypt(int row1, int col1, int row2, int col2){
    if (row1==row2){
        col1=col1-1;
        col2=col2-1;
        if (col1<0)
            col1=4;
```

```
        if (col2<0)
            col2=4;

decrypted+=(Character.toString(alphabets[row1][col1])+Character.toString(alphabets[row1][col2]));

    }else if(col1==col2){
        row1=row1-1;
        row2=row2-1;
        if (row1<0)
            row1=4;
        if (row2<0)
            row2=4;

decrypted+=(Character.toString(alphabets[row1][col1])+Character.toString(alphabets[row2][col1]));

    }else{

decrypted+=(Character.toString(alphabets[row1][col2])+Character.toString(alphabets[row2][col1]));

    }
}

String getEncrypted(){
    return encrypted;
}
String getDecrypted(){
    return decrypted;
}



}
```

Read Playfair Cipher online: https://riptutorial.com/cryptography/topic/8869/playfair-cipher

Credits

S. No	Chapters	Contributors
1	Getting started with cryptography	Community, Maarten Bodewes, Rachel Gallen, Ravindra HV
2	Caesar cipher	Community, Dipesh Poudel, Maarten Bodewes, Ravindra HV
3	Playfair Cipher	Dipesh Poudel