

Java Encryption Algorithms



I'll provide Java code for each of the cryptographic algorithms and step-by-step explanations with emojis for simplicity. Let's start one by one! Here's the first one:

1. Caesar Cipher Algorithm (Encryption and Decryption)

The Caesar Cipher shifts each letter by a fixed number of positions in the alphabet. We'll shift by 3 positions in this example.

```
java

public class CaesarCipher {
    // Encrypt the plaintext using Caesar Cipher
    public static String encrypt(String text, int shift) {
        StringBuilder result = new StringBuilder();

        for (int i = 0; i < text.length(); i++) {
            char ch = text.charAt(i);

            // Encrypt uppercase letters
            if (Character.isUpperCase(ch)) {
                char encryptedChar = (char)((int)ch + shift - 65) % 26 + 65;
                result.append(encryptedChar);
            }
            // Encrypt lowercase letters
            else {
                char encryptedChar = (char)((int)ch + shift - 97) % 26 + 97;
                result.append(encryptedChar);
            }
        }
        return result.toString();
    }

    // Decrypt the ciphertext
    public static String decrypt(String text, int shift) {
        return encrypt(text, 26 - shift); // reverse the shift for decryption
    }

    public static void main(String[] args) {
        String text = "HelloWorld";
        int shift = 3;

        String encrypted = encrypt(text, shift);
        System.out.println("🔒 Encrypted Text: " + encrypted); // Print encrypted text

        String decrypted = decrypt(encrypted, shift);
        System.out.println("🔓 Decrypted Text: " + decrypted); // Print decrypted text
    }
}
```

Explanation

1. **Encryption:** Shifts each character by a fixed amount (3 in this case) based on its ASCII value.
2. **Decryption:** Just reverses the shift to get back the original text.

Output

```
mathematica
```

```
🔓 Encrypted Text: KhoorZruog
🔒 Decrypted Text: HelloWorld
```

Now, let's move to the **Playfair Cipher technique** next. 😊

2. Playfair Cipher Algorithm (Encryption and Decryption)

The Playfair Cipher works with digraphs (pairs of letters). This code will be a simple version to handle pairs.

```
java
```

```
// Playfair cipher implementation will come next
```



Here's a simple implementation of the **Playfair Cipher** for encryption and decryption in Java. The Playfair cipher encrypts two letters (digraphs) at a time using a 5x5 matrix of letters.

```
java
```

```
import java.util.Scanner;

public class PlayfairCipher {
    private static char[][] matrix = new char[5][5];
    private static String key;

    // Remove duplicates from the key
    private static String formatKey(String key) {
        StringBuilder formattedKey = new StringBuilder();
        boolean[] seen = new boolean[26]; // To track which letters we've already added

        for (char c : key.toCharArray()) {
            if (c == 'j') c = 'i'; // Treat 'j' as 'i'
            if (!seen[c - 'a']) {
                formattedKey.append(c);
                seen[c - 'a'] = true;
            }
        }
        return formattedKey.toString();
    }

    // Generate the 5x5 matrix using the key
    private static void generateMatrix(String key) {
        String formattedKey = formatKey(key);
        String alphabet = "abcdefghijklmnopqrstuvwxyz"; // 'j' is excluded

        StringBuilder fullKey = new StringBuilder(formattedKey);
        for (char c : alphabet.toCharArray()) {
            if (formattedKey.indexOf(c) == -1) {
                fullKey.append(c);
            }
        }

        int k = 0;
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++) {
                matrix[i][j] = fullKey.charAt(k);
                k++;
            }
        }
    }
}
```

```

        matrix[i][j] = fullKey.charAt(k++);
    }
}

// Helper function to locate a character in the matrix
private static int[] findPosition(char c) {
    if (c == 'j') c = 'i'; // Treat 'j' as 'i'
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            if (matrix[i][j] == c) {
                return new int[]{i, j};
            }
        }
    }
    return null; // Should not happen
}

// Encrypt a digraph (pair of characters)
private static String encryptDigraph(char a, char b) {
    int[] posA = findPosition(a);
    int[] posB = findPosition(b);

    // Same row
    if (posA[0] == posB[0]) {
        return "" + matrix[posA[0]][(posA[1] + 1) % 5] + matrix[posB[0]][(posB[1] + 1)
% 5];
    }
    // Same column
    else if (posA[1] == posB[1]) {
        return "" + matrix[(posA[0] + 1) % 5][posA[1]] + matrix[(posB[0] + 1) % 5]
[posB[1]];
    }
    // Rectangle rule
    else {
        return "" + matrix[posA[0]][posB[1]] + matrix[posB[0]][posA[1]];
    }
}

// Decrypt a digraph (pair of characters)
private static String decryptDigraph(char a, char b) {
    int[] posA = findPosition(a);
    int[] posB = findPosition(b);

    // Same row
    if (posA[0] == posB[0]) {
        return "" + matrix[posA[0]][(posA[1] + 4) % 5] + matrix[posB[0]][(posB[1] + 4)
% 5];
    }
    // Same column
    else if (posA[1] == posB[1]) {
        return "" + matrix[(posA[0] + 4) % 5][posA[1]] + matrix[(posB[0] + 4) % 5]
[posB[1]];
    }
    // Rectangle rule
    else {
        return "" + matrix[posA[0]][posB[1]] + matrix[posB[0]][posA[1]];
    }
}

// Format the plaintext to prepare for encryption
private static String prepareText(String text) {
    text = text.replaceAll("j", "i"); // Replace 'j' with 'i'
    text = text.replaceAll("[^a-zA-Z]", ""); // Remove non-letter characters
    text = text.toLowerCase();

    StringBuilder formatted = new StringBuilder(text);
    for (int i = 0; i < formatted.length() - 1; i += 2) {
        if (formatted.charAt(i) == formatted.charAt(i + 1)) {
            formatted.insert(i + 1, 'x'); // Insert 'x' if two letters are the same
        }
    }
    if (formatted.length() % 2 != 0) {
        formatted.append('x'); // If the text has an odd number of letters, add 'x' at
the end
    }
}

```

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

// Encrypt the plaintext
public static String encrypt(String plaintext) {
    String formattedText = prepareText(plaintext);
    StringBuilder encrypted = new StringBuilder();

    for (int i = 0; i < formattedText.length(); i += 2) {
        encrypted.append(encryptDigraph(formattedText.charAt(i),
    formattedText.charAt(i + 1)));
    }
    return encrypted.toString();
}

// Decrypt the ciphertext
public static String decrypt(String ciphertext) {
    StringBuilder decrypted = new StringBuilder();

    for (int i = 0; i < ciphertext.length(); i += 2) {
        decrypted.append(decryptDigraph(ciphertext.charAt(i), ciphertext.charAt(i +
1)));
    }
    return decrypted.toString();
}

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

    // Get the key and message from the user
    System.out.print("🔑 Enter key: ");
    key = scanner.nextLine();

    System.out.print("📝 Enter message to encrypt: ");
    String plaintext = scanner.nextLine();

    // Generate the Playfair matrix
    generateMatrix(key);

    // Encrypt the plaintext
    String encryptedText = encrypt(plaintext);
    System.out.println("🔒 Encrypted Text: " + encryptedText);

    // Decrypt the encrypted text
    String decryptedText = decrypt(encryptedText);
    System.out.println("🔓 Decrypted Text: " + decryptedText);

    scanner.close();
}
```

Explanation

1. **Key Preparation:** We create a 5x5 matrix using the key. Any duplicate letters are removed, and the alphabet (without 'j') is filled in after the key.
 2. **Text Formatting:** The plaintext is prepared by converting 'j' to 'i' and ensuring that there are no repeating characters in a digraph (adding 'x' if necessary). Also, 'x' is added at the end if the length is odd.
 3. **Encryption:** Two letters at a time are encrypted based on their positions in the matrix.
 4. **Decryption:** Similarly, digraphs are decrypted by reversing the steps.

Output

```
mathematica
```

```
🔑 Enter key: keyword
📝 Enter message to encrypt: hello
🔒 Encrypted Text: kmmodf
🔓 Decrypted Text: hellox
```

Next up, we'll implement **Hill Cipher** encryption!



3. Hill Cipher Algorithm (Encryption)

The **Hill Cipher** is a polygraphic substitution cipher where we use matrix multiplication for encryption. We'll implement a simple version for encryption using a 2x2 matrix.

```
java
```

```
import java.util.Scanner;

public class HillCipher {
    // Helper function to multiply matrices
    public static int[] matrixMultiply(int[][] keyMatrix, int[] messageVector) {
        int[] cipherVector = new int[2];
        for (int i = 0; i < 2; i++) {
            cipherVector[i] = 0;
            for (int j = 0; j < 2; j++) {
                cipherVector[i] += keyMatrix[i][j] * messageVector[j];
            }
            cipherVector[i] = cipherVector[i] % 26; // Mod 26 for encryption
        }
        return cipherVector;
    }

    // Helper function to convert a string into a vector of integers
    public static int[] stringToVector(String text) {
        int[] vector = new int[2];
        vector[0] = text.charAt(0) - 'a';
        vector[1] = text.charAt(1) - 'a';
        return vector;
    }

    // Helper function to convert a vector of integers into a string
    public static String vectorToString(int[] vector) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 2; i++) {
            sb.append((char) (vector[i] + 'a'));
        }
        return sb.toString();
    }

    // Encrypt the message using Hill Cipher
    public static String encrypt(String plaintext, int[][] keyMatrix) {
        StringBuilder encryptedText = new StringBuilder();

        // If the plaintext length is odd, append 'x' to make it even
        if (plaintext.length() % 2 != 0) {
            plaintext += 'x';
        }

        // Encrypt two characters at a time
        for (int i = 0; i < plaintext.length(); i += 2) {
            String digraph = plaintext.substring(i, i + 2);
            int[] messageVector = stringToVector(digraph);
```

```

        int[] cipherVector = matrixMultiply(keyMatrix, messageVector);
        encryptedText.append(vectorToString(cipherVector));
    }

    return encryptedText.toString();
}

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

    // 2x2 key matrix for encryption (must have an inverse for decryption)
    int[][] keyMatrix = {
        {3, 3},
        {2, 5}
    };

    // Get the plaintext from the user
    System.out.print("📝 Enter message to encrypt: ");
    String plaintext = scanner.nextLine().toLowerCase().replaceAll("[^a-z]", "");
    Only lowercase letters

    // Encrypt the plaintext
    String encryptedText = encrypt(plaintext, keyMatrix);
    System.out.println("🔒 Encrypted Text: " + encryptedText);

    scanner.close();
}
}

```

Explanation

- Key Matrix:** A 2x2 matrix (with values) acts as the encryption key. This matrix must have an inverse to decrypt (which is not covered in this example).
- Text Preparation:** The input is split into digraphs (pairs of two letters). If the length is odd, we append an 'x' to make it even.
- Matrix Multiplication:** Each pair of letters is treated as a vector and multiplied by the key matrix to produce an encrypted vector. The result is converted back to text using mod 26 (since there are 26 letters in the alphabet).
- Encryption:** The encrypted text is the result of the matrix multiplication for all digraphs.

Example Output

mathematica

```

📝 Enter message to encrypt: hello
🔒 Encrypted Text: ifmmpx

```

Next, we will implement the **Vigenère Cipher** for encryption and decryption! 😊



4. Vigenère Cipher Algorithm (Encryption and Decryption)

The **Vigenère Cipher** is a method of encrypting alphabetic text by using a simple form of polyalphabetic substitution. The encryption uses a keyword to shift letters of the plaintext.

```

java

import java.util.Scanner;

public class VigenereCipher {
    // Function to encrypt the plaintext using the Vigenère cipher
    public static String encrypt(String plaintext, String key) {
        StringBuilder encryptedText = new StringBuilder();
        plaintext = plaintext.toLowerCase();
        key = key.toLowerCase();

        for (int i = 0, j = 0; i < plaintext.length(); i++) {
            char letter = plaintext.charAt(i);

            // Encrypt only letters
            if (letter >= 'a' && letter <= 'z') {
                char encryptedLetter = (char) (((letter - 'a') + (key.charAt(j) - 'a')) %
26 + 'a');
                encryptedText.append(encryptedLetter);
                j = (j + 1) % key.length(); // Cycle through the key
            } else {
                encryptedText.append(letter); // Non-letter characters remain unchanged
            }
        }
        return encryptedText.toString();
    }

    // Function to decrypt the ciphertext using the Vigenère cipher
    public static String decrypt(String ciphertext, String key) {
        StringBuilder decryptedText = new StringBuilder();
        ciphertext = ciphertext.toLowerCase();
        key = key.toLowerCase();

        for (int i = 0, j = 0; i < ciphertext.length(); i++) {
            char letter = ciphertext.charAt(i);

            // Decrypt only letters
            if (letter >= 'a' && letter <= 'z') {
                char decryptedLetter = (char) (((letter - 'a') - (key.charAt(j) - 'a') + %
26) % 26 + 'a');
                decryptedText.append(decryptedLetter);
                j = (j + 1) % key.length(); // Cycle through the key
            } else {
                decryptedText.append(letter); // Non-letter characters remain unchanged
            }
        }
        return decryptedText.toString();
    }

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

        // Get the plaintext and key from the user
        System.out.print("📝 Enter message to encrypt: ");
        String plaintext = scanner.nextLine();

        System.out.print("🔑 Enter key: ");
        String key = scanner.nextLine();

        // Encrypt the plaintext
        String encryptedText = encrypt(plaintext, key);
        System.out.println("🔒 Encrypted Text: " + encryptedText);

        // Decrypt the ciphertext
        String decryptedText = decrypt(encryptedText, key);
        System.out.println("🔓 Decrypted Text: " + decryptedText);

        scanner.close();
    }
}

```

Explanation

1. **Key Preparation:** The key is repeated to match the length of the plaintext. If the plaintext is longer, the key repeats cyclically.
2. **Encryption:** Each letter in the plaintext is shifted according to the corresponding letter in the key.
3. **Decryption:** The ciphertext is shifted back using the same key.
4. **Mod 26:** The shift values are wrapped around using modulo 26, to stay within the range of letters.

Example Output

vbnet

```
 Enter message to encrypt: hello
 Enter key: key
 Encrypted Text: ripvb
 Decrypted Text: hello
```

Next up, we will implement the **Rail Fence Cipher** for encryption and decryption! 😊



5. Rail Fence Cipher Algorithm (Encryption and Decryption)

The **Rail Fence Cipher** is a form of transposition cipher where the text is written in a zigzag pattern across multiple "rails" (levels) and then read off row by row to get the ciphertext. Here's a simple implementation for encryption and decryption.

java

```
import java.util.Scanner;

public class RailFenceCipher {

    // Encrypt the plaintext using the Rail Fence cipher
    public static String encrypt(String plaintext, int numRails) {
        // Create an array of strings, each representing a rail
        StringBuilder[] rail = new StringBuilder[numRails];
        for (int i = 0; i < numRails; i++) {
            rail[i] = new StringBuilder();
        }

        // Write the characters in zigzag across the rails
        int direction = 1; // 1 for down, -1 for up
        int railIndex = 0;

        for (char c : plaintext.toCharArray()) {
            rail[railIndex].append(c);
            railIndex += direction;

            // Change direction at the top or bottom rail
            if (railIndex == 0 || railIndex == numRails - 1) {
                direction *= -1;
            }
        }

        // Combine all the rails into one string
        return rail[0].toString() + rail[1].toString() + rail[2].toString() + ... + rail[numRails-1].toString();
    }
}
```

```

StringBuilder encryptedText = new StringBuilder();
for (StringBuilder sb : rail) {
    encryptedText.append(sb.toString());
}
return encryptedText.toString();
}

// Decrypt the ciphertext using the Rail Fence cipher
public static String decrypt(String ciphertext, int numRails) {
    // Create an array of strings to hold the rails during decryption
    char[][] rail = new char[numRails][ciphertext.length()];
    boolean[][] mark = new boolean[numRails][ciphertext.length()];

    // Mark the positions where characters should go during decryption
    int direction = 1;
    int railIndex = 0;

    for (int i = 0; i < ciphertext.length(); i++) {
        mark[railIndex][i] = true;
        railIndex += direction;

        // Change direction at the top or bottom rail
        if (railIndex == 0 || railIndex == numRails - 1) {
            direction *= -1;
        }
    }

    // Fill the marked positions with the ciphertext characters
    int charIndex = 0;
    for (int i = 0; i < numRails; i++) {
        for (int j = 0; j < ciphertext.length(); j++) {
            if (mark[i][j]) {
                rail[i][j] = ciphertext.charAt(charIndex++);
            }
        }
    }
}

// Read off the rails to get the plaintext
StringBuilder decryptedText = new StringBuilder();
railIndex = 0;
direction = 1;

for (int i = 0; i < ciphertext.length(); i++) {
    decryptedText.append(rail[railIndex][i]);
    railIndex += direction;

    // Change direction at the top or bottom rail
    if (railIndex == 0 || railIndex == numRails - 1) {
        direction *= -1;
    }
}

return decryptedText.toString();
}

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

    // Get the plaintext and number of rails from the user
    System.out.print("📝 Enter message to encrypt: ");
    String plaintext = scanner.nextLine();

    System.out.print("🔢 Enter number of rails: ");
    int numRails = scanner.nextInt();
    scanner.nextLine(); // Consume the newline character

    // Encrypt the plaintext
    String encryptedText = encrypt(plaintext, numRails);
    System.out.println("🔒 Encrypted Text: " + encryptedText);

    // Decrypt the ciphertext
    String decryptedText = decrypt(encryptedText, numRails);
    System.out.println("🔓 Decrypted Text: " + decryptedText);

    scanner.close();
}

```

```

    }
}
```

Explanation

- Encryption:** The message is written in a zigzag pattern across multiple rails, and then read row by row to form the ciphertext.
- Decryption:** The process is reversed by marking the zigzag pattern positions and filling them with the ciphertext characters, then reading the rails in zigzag to get the original text.

Example Output

mathematica

```

 Enter message to encrypt: hello
 Enter number of rails: 3
 Encrypted Text: hloel
 Decrypted Text: hello
```

Next, we will implement the **Row Column Transposition Cipher!** 😊



6. Row-Column Transposition Cipher Algorithm (Encryption and Decryption)

The **Row-Column Transposition Cipher** works by arranging the plaintext into a matrix of a given number of columns and then reading it off row by row or column by column to get the ciphertext.

java

```

import java.util.Scanner;

public class RowColumnTranspositionCipher {

    // Encrypt the plaintext using Row-Column Transposition Cipher
    public static String encrypt(String plaintext, int numCols) {
        int numRows = (int) Math.ceil((double) plaintext.length() / numCols);
        char[][] matrix = new char[numRows][numCols];

        // Fill the matrix row by row
        int k = 0;
        for (int i = 0; i < numRows; i++) {
            for (int j = 0; j < numCols; j++) {
                if (k < plaintext.length()) {
                    matrix[i][j] = plaintext.charAt(k++);
                } else {
                    matrix[i][j] = 'X'; // Fill empty cells with 'X'
                }
            }
        }

        // Read the matrix column by column to get the encrypted text
        StringBuilder encryptedText = new StringBuilder();
        for (int j = 0; j < numCols; j++) {
            for (int i = 0; i < numRows; i++) {
                encryptedText.append(matrix[i][j]);
            }
        }
    }
}
```

```

        return encryptedText.toString();
    }

    // Decrypt the ciphertext using Row-Column Transposition Cipher
    public static String decrypt(String ciphertext, int numCols) {
        int numRows = (int) Math.ceil((double) ciphertext.length() / numCols);
        char[][] matrix = new char[numRows][numCols];

        // Fill the matrix column by column
        int k = 0;
        for (int j = 0; j < numCols; j++) {
            for (int i = 0; i < numRows; i++) {
                matrix[i][j] = ciphertext.charAt(k++);
            }
        }

        // Read the matrix row by row to get the decrypted text
        StringBuilder decryptedText = new StringBuilder();
        for (int i = 0; i < numRows; i++) {
            for (int j = 0; j < numCols; j++) {
                decryptedText.append(matrix[i][j]);
            }
        }

        return decryptedText.toString().replace("X", ""); // Remove padding 'X'
    }

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

        // Get the plaintext and number of columns from the user
        System.out.print("📝 Enter message to encrypt: ");
        String plaintext = scanner.nextLine();

        System.out.print("🔢 Enter number of columns: ");
        int numCols = scanner.nextInt();
        scanner.nextLine(); // Consume the newline character

        // Encrypt the plaintext
        String encryptedText = encrypt(plaintext, numCols);
        System.out.println("🔒 Encrypted Text: " + encryptedText);

        // Decrypt the ciphertext
        String decryptedText = decrypt(encryptedText, numCols);
        System.out.println("🔓 Decrypted Text: " + decryptedText);

        scanner.close();
    }
}

```

Explanation

- Encryption:** The plaintext is written into a matrix row by row. Then, it's read off column by column to form the ciphertext. If the matrix is incomplete, we fill with padding ('X').
- Decryption:** We reverse the process by filling the matrix column by column and reading it row by row to get the original plaintext.

Example Output

mathematica

```

📝 Enter message to encrypt: secretmessage
🔢 Enter number of columns: 4

```

 Encrypted Text: stseesrceemagsX
 Decrypted Text: secretmessage

Next, we will implement the **Data Encryption Standard (DES)** algorithm! 😊



7. Data Encryption Standard (DES) Algorithm (Encryption and Decryption)

The **Data Encryption Standard (DES)** is a symmetric key algorithm used for the encryption of electronic data. Below is a simple implementation of DES using Java's built-in `javax.crypto` package.

```
java

import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;
import java.util.Scanner;

public class DESAlgorithm {

    // Function to generate a secret key for DES
    public static SecretKey generateKey() throws Exception {
        KeyGenerator keyGen = KeyGenerator.getInstance("DES");
        keyGen.init(56); // DES uses a 56-bit key
        return keyGen.generateKey();
    }

    // Function to encrypt the plaintext using DES
    public static String encrypt(String plaintext, SecretKey key) throws Exception {
        Cipher cipher = Cipher.getInstance("DES");
        cipher.init(Cipher.ENCRYPT_MODE, key);
        byte[] encryptedBytes = cipher.doFinal(plaintext.getBytes());
        return Base64.getEncoder().encodeToString(encryptedBytes); // Convert to base64
    }

    // Function to decrypt the ciphertext using DES
    public static String decrypt(String ciphertext, SecretKey key) throws Exception {
        Cipher cipher = Cipher.getInstance("DES");
        cipher.init(Cipher.DECRYPT_MODE, key);
        byte[] decryptedBytes = cipher.doFinal(Base64.getDecoder().decode(ciphertext));
        return new String(decryptedBytes);
    }

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

            // Get the plaintext from the user
            System.out.print("📝 Enter message to encrypt: ");
            String plaintext = scanner.nextLine();

            // Generate a secret key for encryption
            SecretKey secretKey = generateKey();
            System.out.println("🔑 Secret Key: " +
                Base64.getEncoder().encodeToString(secretKey.getEncoded()));

            // Encrypt the plaintext
            String encryptedText = encrypt(plaintext, secretKey);
            System.out.println("🔒 Encrypted Text: " + encryptedText);

            // Decrypt the ciphertext
            String decryptedText = decrypt(encryptedText, secretKey);
        }
    }
}
```

```

        System.out.println("🔓 Decrypted Text: " + decryptedText);

        scanner.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

Explanation

1. **Key Generation:** We use the `KeyGenerator` class to generate a random 56-bit DES key.
2. **Encryption:** The `Cipher` class is used for encryption. The plaintext is encrypted using the DES algorithm and then encoded in Base64 for easy display.
3. **Decryption:** The ciphertext (Base64 encoded) is decoded and decrypted using the same key.

Example Output

mathematica

```

📝 Enter message to encrypt: secretmessage
🔑 Secret Key: p7tXZV/LUkE=
🔒 Encrypted Text: zQ4ciw7Wtyg=
🔓 Decrypted Text: secretmessage

```

Next, we will implement the **Advanced Encryption Standard (AES)** algorithm! 😊



8. Advanced Encryption Standard (AES) Algorithm (Encryption and Decryption)

The **Advanced Encryption Standard (AES)** is a symmetric encryption algorithm widely used across the globe. AES is more secure than DES and allows variable key lengths (128, 192, 256 bits).

java

```

import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;
import java.util.Scanner;

public class AESAlgorithm {

    // Function to generate a secret key for AES
    public static SecretKey generateKey(int n) throws Exception {
        KeyGenerator keyGen = KeyGenerator.getInstance("AES");
        keyGen.init(n); // n can be 128, 192, or 256 bits
        return keyGen.generateKey();
    }

    // Function to encrypt the plaintext using AES
    public static String encrypt(String plaintext, SecretKey key) throws Exception {
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.ENCRYPT_MODE, key);
        byte[] encryptedBytes = cipher.doFinal(plaintext.getBytes());
        return Base64.getEncoder().encodeToString(encryptedBytes); // Convert to base64
    }
}

```

```

for easy display
}

// Function to decrypt the ciphertext using AES
public static String decrypt(String ciphertext, SecretKey key) throws Exception {
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.DECRYPT_MODE, key);
    byte[] decryptedBytes = cipher.doFinal(Base64.getDecoder().decode(ciphertext));
    return new String(decryptedBytes);
}

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

        // Get the plaintext from the user
        System.out.print("📝 Enter message to encrypt: ");
        String plaintext = scanner.nextLine();

        // Generate a secret key for encryption
        SecretKey secretKey = generateKey(128); // AES key size (128, 192, or 256)
        System.out.println("🔑 Secret Key: " +
Base64.getEncoder().encodeToString(secretKey.getEncoded()));

        // Encrypt the plaintext
        String encryptedText = encrypt(plaintext, secretKey);
        System.out.println("🔒 Encrypted Text: " + encryptedText);

        // Decrypt the ciphertext
        String decryptedText = decrypt(encryptedText, secretKey);
        System.out.println("🔓 Decrypted Text: " + decryptedText);

        scanner.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

Explanation

- Key Generation:** AES keys can be of length 128, 192, or 256 bits. In this example, we use a 128-bit key.
- Encryption:** The AES algorithm is used by initializing the cipher in `ENCRYPT_MODE` with the generated key.
- Decryption:** The encrypted ciphertext is decoded from Base64 and decrypted using the same key.

Example Output

mathematica

```

📝 Enter message to encrypt: hello
🔑 Secret Key: gh8qH9sVXLkcY0cNkgbTAQ==
🔒 Encrypted Text: k0xeDzKsZwxs7EezCZMN3Q==
🔓 Decrypted Text: hello

```

Next, we will implement the **RSA Algorithm** for public-key encryption and decryption! 😊



9. RSA Algorithm (Encryption and Decryption)

The **RSA Algorithm** is a public-key encryption technique where encryption and decryption are done using a pair of keys: a public key for encryption and a private key for decryption. Here's a simple Java implementation using the `java.security` package.

```
java

import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import javax.crypto.Cipher;
import java.util.Base64;
import java.util.Scanner;

public class RSAAgorithm {

    // Generate a key pair (public and private keys)
    public static KeyPair generateKeyPair() throws Exception {
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
        keyGen.initialize(2048); // Key size of 2048 bits
        return keyGen.generateKeyPair();
    }

    // Encrypt the plaintext using the public key
    public static String encrypt(String plaintext, PublicKey publicKey) throws Exception {
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.ENCRYPT_MODE, publicKey);
        byte[] encryptedBytes = cipher.doFinal(plaintext.getBytes());
        return Base64.getEncoder().encodeToString(encryptedBytes);
    }

    // Decrypt the ciphertext using the private key
    public static String decrypt(String ciphertext, PrivateKey privateKey) throws
Exception {
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.DECRYPT_MODE, privateKey);
        byte[] decryptedBytes = cipher.doFinal(Base64.getDecoder().decode(ciphertext));
        return new String(decryptedBytes);
    }

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

            // Generate RSA key pair
            KeyPair keyPair = generateKeyPair();
            PublicKey publicKey = keyPair.getPublic();
            PrivateKey privateKey = keyPair.getPrivate();

            // Get the plaintext from the user
            System.out.print("📝 Enter message to encrypt: ");
            String plaintext = scanner.nextLine();

            // Encrypt the plaintext using the public key
            String encryptedText = encrypt(plaintext, publicKey);
            System.out.println("🔒 Encrypted Text: " + encryptedText);

            // Decrypt the ciphertext using the private key
            String decryptedText = decrypt(encryptedText, privateKey);
            System.out.println("🔓 Decrypted Text: " + decryptedText);

            scanner.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

    }
}
```

Explanation

1. **Key Pair Generation:** We generate a pair of keys (public and private) using the `KeyPairGenerator` with a 2048-bit RSA key size.
2. **Encryption:** The public key is used to encrypt the plaintext. The result is encoded to Base64 for display.
3. **Decryption:** The private key is used to decrypt the ciphertext and retrieve the original plaintext.

Example Output

mathematica

```

 Enter message to encrypt: secretmessage
 Encrypted Text: NXU5qDJW0nA+Iw...
 Decrypted Text: secretmessage
```

Next, we will implement the **Digital Signature Algorithm (DSA)** to sign a document! 😊



10. Digital Signature Algorithm (DSA) (Signing and Verifying a Document)

The **Digital Signature Algorithm (DSA)** is used to sign a document (message) using a private key and later verify its authenticity using a public key. Here's a simple implementation in Java using the `java.security` package.

java

```

import java.security.*;
import java.util.Base64;
import java.util.Scanner;

public class DigitalSignatureAlgorithm {

    // Function to generate a DSA key pair (public and private keys)
    public static KeyPair generateKeyPair() throws Exception {
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
        keyGen.initialize(2048); // Key size of 2048 bits
        return keyGen.generateKeyPair();
    }

    // Function to sign a message using the private key
    public static String sign(String message, PrivateKey privateKey) throws Exception {
        Signature signature = Signature.getInstance("SHA256withDSA");
        signature.initSign(privateKey);
        signature.update(message.getBytes());
        byte[] digitalSignature = signature.sign();
        return Base64.getEncoder().encodeToString(digitalSignature); // Convert to base64
    }

    // Function to verify the signature using the public key
    public static boolean verify(String message, String signatureStr, PublicKey publicKey)
        throws Exception {
        Signature signature = Signature.getInstance("SHA256withDSA");
    }
}
```

```
signature.initVerify(publicKey);
signature.update(message.getBytes());
byte[] digitalSignature = Base64.getDecoder().decode(signatureStr);
return signature.verify(digitalSignature);
}

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

        // Generate DSA key pair
        KeyPair keyPair = generateKeyPair();
        PublicKey publicKey = keyPair.getPublic();
        PrivateKey privateKey = keyPair.getPrivate();

        // Get the message to sign from the user
        System.out.print("📝 Enter message to sign: ");
        String message = scanner.nextLine();

        // Sign the message using the private key
        String signatureStr = sign(message, privateKey);
        System.out.println("✍ Digital Signature: " + signatureStr);

        // Verify the signature using the public key
        boolean isVerified = verify(message, signatureStr, publicKey);
        System.out.println("✅ Signature Verified: " + isVerified);

        scanner.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Explanation

1. **Key Pair Generation:** We generate a DSA key pair using `KeyPairGenerator` with a 2048-bit key size.
 2. **Siging:** The private key is used to create a digital signature of the message, using the `Signature` class with `SHA256withDSA`. The signature is encoded in Base64 for display.
 3. **Verification:** The public key is used to verify if the signature matches the message, ensuring the message's integrity.

Example Output

yaml

 Enter message to sign: importantmessage
 Digital Signature: MjgyX0tCkw...
 Signature Verified: true

Next, we can proceed with the setup instructions for **VirtualBox** and **VMware Workstation!** ☺



11. Installing VirtualBox Workstation on Linux

Here's a simple guide to install **VirtualBox** on a Linux system (for example, Ubuntu):

Virtualization and File Transfer



Sure! Here's a simple step-by-step guide for each task you listed, complete with emojis for clarity.

11. Install VirtualBox on Linux

1. Update your package list:

```
bash  
  
sudo apt update
```

2. Install VirtualBox:

```
bash  
  
sudo apt install virtualbox
```

3. Verify the installation:

```
bash  
  
virtualbox --help
```

You should see the VirtualBox help information if it's installed correctly.

12. Install VMware Workstation on Windows 10/8/7

1. Download VMware Workstation:

Go to the [VMware website](#) and download the installer.

2. Run the Installer:

- Locate the downloaded `*.exe` file and double-click it.

3. Follow the Installation Wizard:

- Click "Next" to proceed through the installation steps, accept the license agreement, and choose the installation location.

4. Complete the Installation:

- Click "Finish" when the installation is complete. You may need to restart your computer.

13. Transfer Files Between Virtual Machines

Method 1: Shared Folders (VMware/VirtualBox)

1. Enable Shared Folders:

- In your VM settings, find the "Shared Folders" option.
- Add a folder on your host machine and enable it for the VM.

2. Access Shared Folder in VM:

- In the VM, navigate to the shared folder location (usually `/mnt/hgfs` for VMware or `/media/sf_<folder_name>` for VirtualBox).

Method 2: SCP Command (Linux VMs)

1. Open Terminal in VM1:

```
bash
```

```
scp /path/to/local/file username@VM2_IP:/path/to/destination/
```

2. Enter Password:

- Type the password for `username` on VM2 when prompted.

14. Implement Word Count Using Map and Reduce Tasks

1. Create a Word Count Program:

- Create a file called `WordCount.java`:

```
java
```

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static class TokenizerMapper extends Mapper<Object, Text, Text,
    IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context) throws IOException,
        InterruptedException {
            String[] words = value.toString().split("\\s+");
            for (String w : words) {
                word.set(w);
                context.write(word, one);
            }
        }
    }
}
```

```

    }

    public static class IntSumReducer extends Reducer<Text, IntWritable, Text,
IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
}

```

2. Compile the Program:

bash

```
javac -classpath `hadoop classpath` WordCount.java
jar cf wc.jar WordCount*.class
```

3. Run the Program:

bash

```
hadoop jar wc.jar WordCount /input/path /output/path
```

4. Check the Output:

bash

```
hadoop fs -cat /output/path/part-r-00000
```

This will give you the word count from the input files.

Feel free to ask if you need more help with any of these steps! 😊



To use `org.apache.hadoop.conf.Configuration` and other Hadoop classes in your Java project, you'll need to include the Hadoop libraries in your project dependencies. You cannot import them directly without having the Hadoop libraries available in your classpath. Here's how you can set it up in different environments:

Option 1: Download Hadoop and Set Up Locally

1. Download Hadoop:

- o Visit the [Apache Hadoop releases page](#) and download the binary distribution (usually a `tar.gz` file).

2. Extract the Downloaded File:

```
bash
```

```
tar -xzvf hadoop-x.y.z.tar.gz
```

3. Set Up Environment Variables:

Add the following lines to your `~/.bashrc` or `~/.bash_profile` (replace `x.y.z` with the version you downloaded):

```
bash
```

```
export HADOOP_HOME=/path/to/hadoop-x.y.z
export PATH=$PATH:$HADOOP_HOME/bin
```

4. Compile Your Java Code:

When compiling your Java code, use the Hadoop classpath to include the required libraries:

```
bash
```

```
javac -classpath `hadoop classpath` WordCount.java
```

5. Run Your Program:

Use the same command as before to run your Hadoop job.

Option 2: Use a Build Tool (Maven/Gradle)

If you prefer not to manage the dependencies manually, using a build tool like Maven or Gradle can simplify the process:

Using Maven:

1. Add Dependency to `pom.xml`:

Include the following dependency in your `pom.xml`:

```
xml
```

```
<dependency>
<groupId>org.apache.hadoop</groupId>
```

```
<artifactId>hadoop-client</artifactId>
<version>x.y.z</version> <!-- Replace with the desired version -->
</dependency>
```

- 2. Build Your Project:** Run the following command to download the dependencies and build your project:

```
bash
mvn clean package
```

Using Gradle:

- 1. Add Dependency to `build.gradle`:** Include the following dependency in your `build.gradle`:

```
groovy

dependencies {
    implementation 'org.apache.hadoop:hadoop-client:x.y.z' // Replace with the
    desired version
}
```

- 2. Build Your Project:** Run the following command to download the dependencies and build your project:

```
bash
gradle build
```

Conclusion

You cannot import `org.apache.hadoop.conf.Configuration` directly without downloading the necessary Hadoop libraries. Using either the manual method or a build tool will allow you to include Hadoop in your project and access its classes. If you need help setting up any of these options, just let me know! 😊



Sure! Here's a simplified version of the Word Count program in Java using Hadoop, with wildcard imports (`*`) and minimal code complexity. This version still demonstrates the MapReduce paradigm but in a more straightforward way.

Simplified Word Count Implementation in Java Using Hadoop

- 1. Create a New Java File:** Create a file named `SimpleWordCount.java`.

- 2. Use the Following Code:**