```java
import java.io.*;
import java.math.BigInteger;
import java.net.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import java.nio.charset.StandardCharsets;

public class DHServer {

    public static void main(String[] args) throws IOException, NoSuchAlgorithmException, NoSuchPaddingException,
            InvalidKeyException, IllegalBlockSizeException, BadPaddingException, ClassNotFoundException,
            InvalidAlgorithmParameterException {
//The public static void main(String[] args) method is the entry point of the Java program. It's the starting point of execution when you run the Java application. Let's break down the
// public: This keyword is an access modifier that indicates the visibility of the method. In this case, public means that the main method can be accessed from outside the class.
// static: This keyword indicates that the main method belongs to the class itself rather than to an instance of the class. It allows the method to be called without creating an instar
// void: This is the return type of the main method, indicating that it doesn't return any value.
// main: This is the name of the method. The main method is a special method in Java that serves as the entry point for the execution of the program.
// (String[] args): This is the parameter list for the main method. It accepts an array of strings as arguments. When you run a Java program from the command line, you can provide argu
// throws IOException, NoSuchAlgorithmException, NoSuchPaddingException, InvalidKeyException, IllegalBlockSizeException, BadPaddingException, ClassNotFoundException, InvalidAlgorithmPa
// IOException: Signals that an I/O (input/output) exception has occurred.
// NoSuchAlgorithmException: Indicates that a requested cryptographic algorithm is not available.
// NoSuchPaddingException: Indicates that a requested padding scheme is not available.
// InvalidKeyException: Indicates an invalid key has been encountered.
// IllegalBlockSizeException: Indicates that a block cipher operation was attempted with an illegal block size.
// BadPaddingException: Indicates that the padding of a message is incorrect.
// ClassNotFoundException: Thrown when an application tries to load in a class through its string name but no definition for the class with the specified name could be found.
// InvalidAlgorithmParameterException: Indicates that one of the algorithm parameters passed to a method is invalid.
// The main method is a standard convention in Java, and it serves as the starting point for the program's execution. Any code within this method will be executed when the program is r

        System.out.println("#################################################");
        System.out.println("Start by listening on port no 11111");
        System.out.println("#################################################");
//System.out.println(...): This line prints a message to the standard
//output (usually the console). In this case,
//it's printing the message "Start by listening on port no 11111".
        ServerSocket ss = new ServerSocket(11111);
        //ServerSocket: This class is part of Java's networking support and is used to create a server socket, which listens for incoming client connections on a specified port.

// ss: This is a variable that holds
// the reference to the ServerSocket object.

// new ServerSocket(11111): This part creates a new instance of
// ServerSocket and binds it to port number 11111.
// The server will listen for incoming connections on this port.
        System.out.println("#################################################");
        System.out.println("Waiting for client connection...");
        System.out.println("#################################################");
        //Similar to the first System.out.println(...), this line prints a message to the console.
        //In this case, it's printing "Waiting for client connection...".

        while (true) {
            Socket link = ss.accept();

            // ss.accept(): This method blocks until a client connects
            // to the server socket, and then it returns a new Socket
            // object representing the connection to the client.

            // Socket link: This is a variable that holds the reference
            // to the Socket object representing the connection to the client.
            System.out.println("#################################################");
            System.out.println("Connected to client: " + link.getInetAddress().toString());
            System.out.println("#################################################");

            // This line prints a message to the console indicating that the
            //  server has successfully connected to a client. link.getInetAddress().toString()
            // retrieves the IP address of the connected client.

            BufferedReader in = new BufferedReader(new InputStreamReader(link.getInputStream()));
            PrintStream out = new PrintStream(link.getOutputStream());

        // These lines set up input and output streams
        // for communication with the connected client. link.getInputStream()
        // gets the input stream associated with the socket,
        //and link.getOutputStream() gets the output stream.

        // BufferedReader is used to read
        // text from a character-based input stream,
        // and PrintStream is used to print formatted
        // representations of objects to a text-output stream.

            BigInteger q = new BigInteger(in.readLine());
            BigInteger a = new BigInteger(in.readLine());
            BigInteger ya = new BigInteger(in.readLine());

            // These lines read three BigInteger values from the
            // client through the input stream. The assumption is that
            //  the client sends these values, likely as part of a
            //   Diffie-Hellman key exchange.

            SecureRandom sr = new SecureRandom();
            BigInteger xb = new BigInteger(q.bitLength() - 1, sr);
            BigInteger yb = a.modPow(xb, q);

            out.println(yb);

            // These lines generate a random BigInteger (xb) of the same
            // bit length as q and then computes yb using modular
            // exponentiation. The result is sent back to the client
            // through the output stream.

            BigInteger key = ya.modPow(xb, q);

            System.out.println("#################################################");
            System.out.println("The secret key with " + link.getInetAddress().toString() + " is\n" + key);
            System.out.println("#################################################");

            // This calculates the shared secret key (key) using modular
            // exponentiation and
            //  prints the result to the console along with the client's IP address.

            BufferedReader userEntry = new BufferedReader(new InputStreamReader(System.in));

            // This sets up a BufferedReader to read input from the user
            //  via the console. It will be
            // used for entering messages or commands during the
            // communication with the client.

            while (true) {
```

```
    // This starts an infinite loop, meaning the code
    // inside this loop will keep running until explicitly broken out of.

    try {
        // The following block is placed inside a
        // try-catch block to handle exceptions that might occur during execution.

        // Receive and decrypt the message
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
         // PKCS5Padding is a padding scheme used in cryptographic
        //  operations, particularly in block cipher modes such as
        //   Cipher Block Chaining (CBC) mode. Padding is necessary when the
        //    length of the data to be encrypted is not a multiple of the block
        //     size of the cipher. In the case of AES (Advanced Encryption Standard),
        //      which has a block size of 128 bits (16 bytes),
        //  PKCS5Padding adds padding to the plaintext so that its length
        //  becomes a multiple of the block size.
        // This line creates a Cipher object for AES encryption
        // with CBC (Cipher Block Chaining) mode and PKCS5Padding. It
        // specifies the transformation algorithm to be used for encryption and decryption.

        byte[] sharedKeyBytes = key.toByteArray();
        SecretKeySpec sharedSecretKey = new SecretKeySpec(sharedKeyBytes, "AES");
        // These lines convert the shared key (key) to a byte array and
        // then create a SecretKeySpec using this byte array.
        // This key specification is needed for initializing the cipher.

        // Add IV (Initialization Vector)
        byte[] ivBytes = new byte[cipher.getBlockSize()];
        IvParameterSpec ivSpec = new IvParameterSpec(ivBytes);

        // Here, an initialization vector (IV) is created.
        // IV is necessary for AES in CBC mode.
        // The size of the IV is determined by the block size of the cipher.

        cipher.init(Cipher.DECRYPT_MODE, sharedSecretKey, ivSpec);
        // The cipher is initialized for decryption
        // using the shared secret key and the IV.

        ObjectInputStream inputStream = new ObjectInputStream(link.getInputStream());
        byte[] encryptedMessage = (byte[]) inputStream.readObject();

        // This reads an object (presumably the encrypted message)
        //  from the input stream of the socket.
        byte[] decryptedMessage = cipher.doFinal(encryptedMessage);
        String message = new String(decryptedMessage, StandardCharsets.UTF_8);
        // The received encrypted message is decrypted using the initialized cipher,
        // and the result is converted to a string using UTF-8 encoding.

        // Display the received encrypted and decrypted messages
        System.out.println("####################################################");
        System.out.println("Received encrypted message: " + new String(encryptedMessage, StandardCharsets.UTF_8));
        System.out.println("####################################################");

        System.out.println("####################################################");
        System.out.println("Decrypted message: " + message);
        System.out.println("####################################################");
        // These lines print both the encrypted and decrypted versions
        // of the received message to the console.
        // The decrypted message is displayed in its human-readable form.

        // Check for exit condition
        if ("exit".equalsIgnoreCase(message)) {
            System.out.println("Client exited.");
            System.out.println("Restarting server for a new connection...");
            link.close();
            in.close();
            out.close();
            break;
        }

        // Input and send a reply message
        System.out.println("####################################################");
        System.out.println("Enter the reply message:");
        System.out.println("####################################################");
        String replyMessage = userEntry.readLine();
        // This prompts the user to enter a reply message in the console.
        // The entered message
        //  is read using userEntry.readLine() and stored in the variable replyMessage.

        // Encrypt and send the reply message
        cipher.init(Cipher.ENCRYPT_MODE, sharedSecretKey, ivSpec);
        byte[] encryptedReply = cipher.doFinal(replyMessage.getBytes(StandardCharsets.UTF_8));

        // The reply message is encrypted using the same cipher instance
        // that was used for decryption. The doFinal method is used
        // to perform the encryption. The result is stored in the encryptedReply byte array.

        ObjectOutputStream outputStream = new ObjectOutputStream(link.getOutputStream());
        outputStream.writeObject(encryptedReply);

        // An ObjectOutputStream is created to write objects to the output
        // stream of the socket.
        //  The encrypted reply message (encryptedReply) is written to the output stream.

        // Display the encrypted version of the reply message
        System.out.println("####################################################");
        System.out.println("Sent encrypted reply: " + new String(encryptedReply, StandardCharsets.UTF_8));
        System.out.println("####################################################");
        // This line prints the encrypted version of the reply message
        //  to the console. The byte array
        // encryptedReply is converted to a string using UTF-8 encoding for display purposes.
    } catch (EOFException e) {
        // This catch block is specifically designed to catch the
        // EOFException that might occur if the client closes the
        // connection unexpectedly. In such a case, it prints a message
        //  and breaks out of the loop,
        //  ending the server's communication with the client.
        System.out.println("####################################################");
        System.out.println("Client connection closed unexpectedly.");
        break;
    }
}

// Close resources for this client
//This marks the end of the while (true) loop.
//If the loop is exited (e.g., due to a client exit),
//the resources associated with the
//current client (socket, input stream, output stream) are closed.
```

```
            // The server continues listening for new client connections in the outer loop.
            link.close();
            in.close();
            out.close();
        }
    }
}
```