Suppose you have a Three step authentication system.

1. First Ipwhitlist authentication: If successful, then move to next authentication(Two Factor Authentication), Otherwise authenticationA fails
2. (If successful) Two factor authentication, if successful, then move to next authentication (Usernname Password Authentication), Otherwise authentication fails
3. (If successful) Username Password authentication ,If successful, then move whole authentication Process successful .Otherwise authentication fails

1

```java
public interface AuthenticationHandler {
    void setNextHandler(AuthenticationHandler authenticationHandler);
    boolean authenticate(String userName, String password);
}
```

2

```java
public class IPWhitelistingHandler implements AuthenticationHandler{
    private AuthenticationHandler authenticationHandler;

    @Override
    public void setNextHandler(AuthenticationHandler authenticationHandler) {
        this.authenticationHandler = authenticationHandler;
    }

    @Override
    public boolean authenticate(String userName, String password) {
        // Simulating IP whitelisting
        String clientIP = getClientIP();
        if (!clientIP.contains("192.168.192.")) {
            System.out.println("IPWhitelistingHandler: Authentication failed.");
            return false;
        } else if (authenticationHandler != null) {
            System.out.println("IPWhitelistingHandler: Authentication successful.");
            return authenticationHandler.authenticate(userName, password);
        } else {
            System.out.println("IPWhitelistingHandler: Authentication failed.");
            return false;
        }
    }

    private String getClientIP() {
        // Simulated method to get client IP address
        return "192.168.192.";
    }
}
```

3

```java
public class TwoFactorAuthenticationHandler implements AuthenticationHandler{
    private AuthenticationHandler authenticationHandler;
    @Override
    public void setNextHandler(AuthenticationHandler authenticationHandler) {
        this.authenticationHandler = authenticationHandler;
    }

    @Override
    public boolean authenticate(String userName, String password) {
        // Simulating two-factor authentication
        if (userName.equals("user") && password.equals("user123") && verifyOTP("123456")) {
            System.out.println("TwoFactorAuthenticationHandler: Authentication successful.");
            return true;
        } else if (authenticationHandler != null) {
            return authenticationHandler.authenticate(userName, password);
        } else {
            System.out.println("TwoFactorAuthenticationHandler: Authentication failed.");
            return false;
        }
    }

    private boolean verifyOTP(String otp) {
        // Simulated OTP verification logic
        return otp.equals("123456");
    }
```

4

```
public class UsernamePasswordAuthenticationHandler implements AuthenticationHandler{
    private AuthenticationHandler authenticationHandler;

    @Override
    public void setNextHandler(AuthenticationHandler authenticationHandler) {
        this.authenticationHandler = authenticationHandler;
    }

    @Override
    public boolean authenticate(String userName, String password) {
        if (userName.equals("admin") && password.equals("admin123")) {
            System.out.println("UsernamePasswordAuthenticationHandler: Authentication successful.");
            return true;
        } else if (authenticationHandler != null) {
            return authenticationHandler.authenticate(userName, password);
        } else {
            System.out.println("UsernamePasswordAuthenticationHandler: Authentication failed.");
            return false;
        }
    }
}
```

5

```
public class Client {
    public static void main(String[] args) {
        AuthenticationHandler upHandler = new UsernamePasswordAuthenticationHandler();
        AuthenticationHandler tfaHandler = new TwoFactorAuthenticationHandler();
        AuthenticationHandler ipHandler = new IPWhitelistingHandler();

        ipHandler.setNextHandler(upHandler);
        upHandler.setNextHandler(tfaHandler);

        boolean isAuthenticated = ipHandler.authenticate("user", "user123");
        if (isAuthenticated) {
            // Proceed with server access
            System.out.println("Access granted.");
        } else {
            // Handle authentication failure
            System.out.println("Access denied.");
        }
    }
}
```

1. We start by defining the `AuthenticationHandler` interface, which represents the base handler in the chain. It has two methods: `setNextHandler()` to set the next handler in the chain and `authenticate()` to perform authentication.
2. Next, we have three concrete implementations of the `AuthenticationHandler` interface: `IPWhitelistingHandler`, `TwoFactorAuthenticationHandler`, and `UsernamePasswordAuthenticationHandler`. Each handler implements the `setNextHandler()` and `authenticate()` methods according to its specif authentication logic.
3. In the `Client` class, we create instances of the authentication handler `upHandler` for username/password authentication, `tfaHandler` for two factor authentication, and `ipHandler` for IP whitelisting.
4. We then set up the chain of responsibility by calling `setNextHandler()` on each handler, in the desired order. In this case, the request will flow from `ipHandler` to `upHandler`, and then to `tfaHandler`.
5. Finally, we call the `authenticate()` method on the `ipHandler` and pass the username and password for authentication. The request will propagate through the chain of handlers until it is handled or reaches the end of the chain.
6. Each handler performs its specific authentication logic and decides whether to handle the request or pass it to the next handler in the chain. If a handler can handle the request, it returns `true`. Otherwise, it delegates the request to the next handler.
7. If the request is handled successfully by any of the handlers, the `isAuthenticated` variable in the `Client` class will be `true`, indicating successful authentication. Otherwise, it will be `false`, indicating authentication failure.
8. Based on the value of `isAuthenticated`, we can proceed with server access if authentication is successful or handle authentication failure accordingly.

1. We start by defining an `Iterator` interface. It declares two methods: `hasNext()` to check if there are more elements, and `next()` to retrieve the next element. This interface serves as a contract for all iterators.
2. Next, we define a `Collection` interface. It declares a single method `getIterator()` that returns an instance of the `Iterator` interface. This interface represents a collection of elements and provides a way to access them using an iterator.
3. We implement the `NameIterator` class, which is a concrete implementation of the `Iterator` interface. It maintains a reference to an array of names (`names`) and a `position` variable to keep track of the current position while iterating. The `hasNext()` method checks if there are more names in the array by comparing the current position with the length of the array. The `next()` method retrieves the next name from the array and increments the position.
4. We implement the `NameCollection` class, which is a concrete implementation of the `Collection` interface. It takes an array of names in its constructor and stores them internally. The `getIterator()` method creates a new instance of the `NameIterator` class and passes the array of names to it. It returns the created iterator, which allows accessing the names in the collection.
5. In the client code (`Main` class), we create an array of names (`names`) containing "John," "Emily," "David," and "Sarah."
6. We create an instance of `NameCollection` called `collection` and pass the `names` array to its constructor. This initializes the collection with the names.
7. We retrieve an iterator from the collection by calling the `getIterator()` method. This gives us an instance of the `NameIterator` class.
8. We use a `while` loop to iterate over the collection. The loop condition checks if the iterator has more elements using the `hasNext()` method.
9. Inside the loop, we retrieve the next name from the iterator using the `next()` method and store it in the `name` variable.
10. Finally, we print each name to the console.

1

```
// Step 1: Iterator interface
interface Iterator {
    boolean hasNext();
    String next();
}
```

2

```
// Step 2: Collection interface
interface Collection {
    Iterator getIterator();
}
```

3

```
// Step 3: NameIterator implementation of Iterator interface
class NameIterator implements Iterator {
    private String[] names;
    private int position;

    public NameIterator(String[] names) {
        this.names = names;
        this.position = 0;
    }

    public boolean hasNext() {
        return position < names.length;
    }

    public String next() {
        String name = names[position];
        position++;
        return name;
    }
}
```

Participant:

1. Iterator
2. IteratableCollection : Collection
3. ConcreteIterator: NameIterator
4. ConcreteCollection : NameCollection

4

```
// Step 4: NameCollection implementation of Collection interface
class NameCollection implements Collection {
    private String[] names;
    public NameCollection(String[] names) {
        this.names = names;
    }
    public Iterator getIterator() {
        return new NameIterator(names);
    }
}
```

5

```
public class Client {
    public static void main(String[] args) {
        String[] names = {"John", "Emily", "David", "Sarah"};

        Collection collection = new NameCollection(names);
        Iterator iterator = collection.getIterator();

        while (iterator.hasNext()) {
            String name = iterator.next();
            System.out.println(name);
        }
    }
}
```