Authentication/Authorization Decorator Pattern: Class Explanations

This document provides a concise overview of each Java class within the `design_patterns.decorator_pattern.authentication_authorization` package, detailing its role in implementing the **Decorator design pattern** for a flexible authentication system.

---

1. `Authenticator` (Interface)

- **Role in Pattern: Component Interface**
- **Purpose:** Establishes the fundamental **contract** for all authentication mechanisms. Both the basic login handler and any added security layers must adhere to this interface.
- **Key Methods:**
    - `authenticate(String username, String password)`: The core operation to process a user's login attempt.
    - `getDescription()`: Provides a string outlining the active authentication method(s).
- **Why it's important:** Ensures that any part of your application can interact with any authenticator (whether simple or complex) in a **uniform way**, without needing to know its internal details.

---

2. `BasicAuthenticator` (Concrete Component)

- **Role in Pattern: Concrete Component**
- **Purpose:** Implements the **most basic form of authentication**, typically a username and password check. This is the foundational object that other security features (decorators) will build upon.
- **Key Features:**
    - Handles the core `authenticate` logic (e.g., verifying credentials).
    - Sets the initial description for the authentication process.
- **Why it's important:** Acts as the **starting point** of the decoration chain, providing the essential authentication functionality before any enhancements are applied.

---

3. `AuthenticatorDecorator` (Abstract Decorator)

- **Role in Pattern: Abstract Decorator**
- **Purpose:** Serves as the **base class for all concrete authentication decorators**. It acts as an intermediary, implementing the `Authenticator` interface itself and holding a reference to another `Authenticator` object (the one it's decorating).
- **Key Features:**
    - Holds a `protected` reference (`authenticator`) to the object it's wrapping.
    - Ensures a valid `Authenticator` is provided through its constructor.

    - Provides **default delegation**: its `authenticate()` and `getDescription()` methods simply pass the call along to the wrapped `authenticator`. Concrete decorators will override these to add their specific logic.
- **Why it's important:** Establishes a **common structure for all decorators** and enables the **chaining mechanism**, allowing security layers to be stacked one upon another.

---

4. `MFAAuthenticator` (Concrete Decorator)

- **Role in Pattern: Concrete Decorator**
- **Purpose:** Adds **Multi-Factor Authentication (MFA)** as an additional layer of security to the authentication flow.
- **Key Features:**
    - Extends `AuthenticatorDecorator`.
    - **Overrides `authenticate()`:** First, it delegates to the wrapped authenticator (e.g., `BasicAuthenticator`). If that succeeds, it then performs its own simulated MFA check. If either fails, authentication stops.
    - **Overrides `getDescription()`:** Appends " with MFA" to the description it gets from the wrapped authenticator, building a more complete description of the security chain.
- **Why it's important:** Demonstrates how to **dynamically add a specific, additive security feature** to the authentication process.

---

5. `RoleBasedAuthAuthenticator` (Concrete Decorator)

- **Role in Pattern: Concrete Decorator**
- **Purpose:** Integrates **role-based authorization** into the authentication flow, ensuring that the authenticated user also possesses a specific, required role for accessing a resource.
- **Key Features:**
    - Extends `AuthenticatorDecorator`.
    - **Overrides `authenticate()`:** Delegates to the wrapped authenticator (which might include MFA). If successful, it then performs a role check based on a `requiredRole` defined for this decorator.
    - **Overrides `getDescription()`:** Appends details about the role check (e.g., " with Role-Based Authentication (Admin)") to the growing description of the authentication chain.
- **Why it's important:** Shows how another distinct **authorization layer** can be dynamically applied and configured, building upon previously established authentication methods.

---

6. `Main` (Demonstration Class)

- **Role in Pattern: Client**
- **Purpose:** Serves as the application's entry point to **demonstrate how to compose and use** the various `Authenticator` components and decorators.
- **Key Features:**
    - Creates instances of `BasicAuthenticator`.
    - **Dynamically wraps** these instances with `MFAAuthenticator` and `RoleBasedAuthAuthenticator` in different combinations, showcasing various security configurations.
    - Invokes the `authenticate()` method on the final decorated object, illustrating how all layered security checks are performed sequentially.
    - Prints the `getDescription()` for each configured authenticator, vividly demonstrating how the description dynamically builds up through composition.
- **Why it's important:** Provides a concrete, runnable example of the **flexibility and power** of the Decorator pattern for runtime feature composition.