Documentation for the Factory Pattern: User Creation

Here's the detailed documentation for the Factory Pattern implementation, which will be highly beneficial for your project viva:

2. Factory Pattern: User Creation

- **Files:**
  - src/main/java/com/ums/User.java (Interface)
  - src/main/java/com/ums/Student.java (Concrete Class)
  - src/main/java/com/ums/Professor.java (Concrete Class)
  - src/main/java/com/ums/Admin.java (Concrete Class)
  - src/main/java/com/ums/UserFactory.java (Factory Class)

Purpose of the Pattern

The Factory Pattern (specifically, a Simple Factory or Static Factory Method) provides an interface for creating objects in a superclass, but allows subclasses (or a dedicated factory class) to instantiate the concrete objects. Its primary goal is to **decouple object creation from object usage**.

In the context of the University Management System (UMS):

- **Centralized Object Creation:** Instead of scattering new Student(), new Professor(), new Admin() calls throughout the application, all user object creation is handled by a single UserFactory.
- **Encapsulation of Creation Logic:** The client code (the part of your application that needs a user object) doesn't need to know the specific class names or the complex logic required to instantiate each user type (e.g., what parameters each constructor takes, or if some setup is needed). It just tells the factory *what* it needs ("Student", "Professor", "Admin"), and the factory handles the *how*.
- **Reduced Client Dependency:** The client code depends only on the User interface and the UserFactory, not on the concrete Student, Professor, or Admin classes. This means changes to how Student objects are created (e.g., adding new constructor parameters) don't require changes in client code, only in the UserFactory.
- **Flexibility and Extensibility:** If a new user type (e.g., Staff or Alumni) needs to be added in the future, you simply create the new concrete class (e.g., Staff.java) and update the UserFactory to handle it. Existing client code that uses the factory **does not need to be modified**. This makes the system much easier to maintain and extend.

How it Works (Implementation Details)

The Factory Pattern is implemented through the following components:

1. **User Interface (User.java):**

Documentation for the Factory Pattern: User Creation

```
Java
public interface User {
    void displayRole();
    String getUserType();
}
```

- This is the product interface. It defines a common contract (methods like displayRole(), getUserType()) that all concrete user types must adhere to. This allows for **polymorphism**, meaning client code can treat any Student, Professor, or Admin object generically as a User.

2. **Concrete Product Classes (Student.java, Professor.java, Admin.java):**

```
Java
// Example: Student.java
public class Student implements User {
    // ... constructor and specific fields ...
    @Override
    public void displayRole() { /* ... */ }
    @Override
    public String getUserType() { return "Student"; }
}
// Similar implementations for Professor and Admin, each with their own specific attributes and display logic.
```

- These are the actual objects created by the factory. Each class implements the User interface, providing their specific characteristics and behavior while fulfilling the common contract.

3. **UserFactory Class (UserFactory.java):**

```
Java
public class UserFactory {
    public static User createUser(String userType, String id, String name, String additionalInfo) {
        // ... logic to create Student, Professor, or Admin based on userType ...
    }
    // Overloaded method:
    public static User createUser(String userType, String id, String name) {
        // ... calls the main createUser with null for additionalInfo ...
    }
}
```

- This is the "factory" itself. It contains one or more static "factory methods" (like createUser()).
- The createUser() method takes parameters (like userType, id, name, additionalInfo) that describe the type of User object needed.

Documentation for the Factory Pattern: User Creation

- Inside this method, a switch statement (or if-else ladder) determines which concrete User class to instantiate based on the userType.
- It then returns a new instance of the appropriate concrete class (e.g., new Student(...), new Professor(...), new Admin(...)).
- Importantly, the method returns a User *interface* type, allowing the caller to use the object polymorphically without needing to know its specific concrete class.
- Includes error handling for unknown user types, making the creation process robust.

Justification for Factory Pattern Implementation

- **Loose Coupling:** The most significant benefit. Client code that needs a User object doesn't need to know about Student, Professor, or Admin classes directly. It just asks the UserFactory for a User, promoting a more modular and maintainable codebase.
- **Single Responsibility Principle:** The UserFactory has a single responsibility: creating user objects. This keeps the user classes focused on their own behavior and attributes, and other parts of the system focused on their own tasks.
- **Maintainability:** If the way users are created changes (e.g., new constructor parameters, or complex initialization), only the UserFactory needs to be updated. The rest of the application using UserFactory.createUser() remains unaffected.
- **Scalability/Extensibility:** Adding new user roles in the future is simple. You create a new class implementing User and add a new case in the UserFactory's createUser method. No other existing code needs to be modified, adhering to the **Open/Closed Principle** (open for extension, closed for modification).
- **Hides Complexity:** The creation logic for different user types (which parameters go where, which constructor to call) is hidden inside the factory, making the client code cleaner and easier to read.

Regarding the [WARNING] thread Thread[#46,mysql-cj-abandoned-connection-cleanup...

This warning is common when running simple Java applications that use JDBC drivers, especially when the application's main method exits quickly after using the database connection. The mysql-cj-abandoned-connection-cleanup thread is a daemon thread started by the MySQL JDBC driver to manage connections and resources. When your main thread finishes, sometimes this cleanup thread hasn't fully shut down yet, leading to the warning.

It is usually not a critical problem for a simple demo application. In a real-world application (like a web server or a long-running service), you would use a connection pool (which would be managed by your Singleton DatabaseConnectionManager in a more advanced setup) that handles the lifecycle of connections and cleanup threads much more robustly. For your current project, this warning can be safely ignored; it doesn't indicate an error in your Singleton or Factory pattern implementations.

Documentation for the Factory Pattern: User Creation

Now that we have both Singleton and Factory patterns well-documented and implemented, we can move to the **Builder Pattern** for constructing complex Course objects!