OOAD ASSIGNMENT

NAME : RAHUL V REDDY

SRN: PES2UG22CS430

SECTION: G

1. Singleton Pattern – Database Connection Manager

Concept: Ensures a single instance of a class, providing a global access point.

Questions:

1. Explain how the Singleton pattern can be used to manage database connections in an application.

   The **Singleton pattern** ensures that only one instance of a class is created and provides a global access point to that instance. In the context of a **Database Connection Manager**, this is useful because:

- It **reduces resource consumption** by ensuring only one database connection is used.
- It **prevents connection leaks** by managing a single instance.
- It **ensures thread safety** when multiple parts of the application access the database.

2. Implement a DatabaseConnection class using the Singleton pattern. Ensure that only one connection instance is created.

```
import sqlite3

class DatabaseConnection:

    _instance = None  # Class-level variable to store the single instance

    def __new__(cls, db_name="database.db"):

        if cls._instance is None:

            cls._instance = super(DatabaseConnection, cls).__new__(cls)

            cls._instance.connection = sqlite3.connect(db_name)  # Open database connection

        return cls._instance


    def get_connection(self):

        return self.connection
```

```
# Testing Singleton Behavior
db1 = DatabaseConnection()
db2 = DatabaseConnection()
print(db1 is db2)  # True (Same instance)
```

3. Modify the DatabaseConnection class to support lazy initialization.

```
import sqlite3
from threading import Lock

class DatabaseConnection:
    _instance = None
    _lock = Lock()  # To ensure thread safety

    def __new__(cls):
        if cls._instance is None:
            with cls._lock:  # Ensures thread safety
                if cls._instance is None:
                    cls._instance = super(DatabaseConnection, cls).__new__(cls)
                    cls._instance.connection = None
        return cls._instance

    def connect(self, db_name="database.db"):
        if self.connection is None:
            self.connection = sqlite3.connect(db_name)  # Create connection only when needed
        return self.connection

# Testing Lazy Initialization
db1 = DatabaseConnection()
conn1 = db1.connect()
db2 = DatabaseConnection()
conn2 = db2.connect()
print(conn1 is conn2)  # True (Same connection)
```

4. What are the potential issues with using the Singleton pattern in a multi-threaded environment, and how can they be resolved?

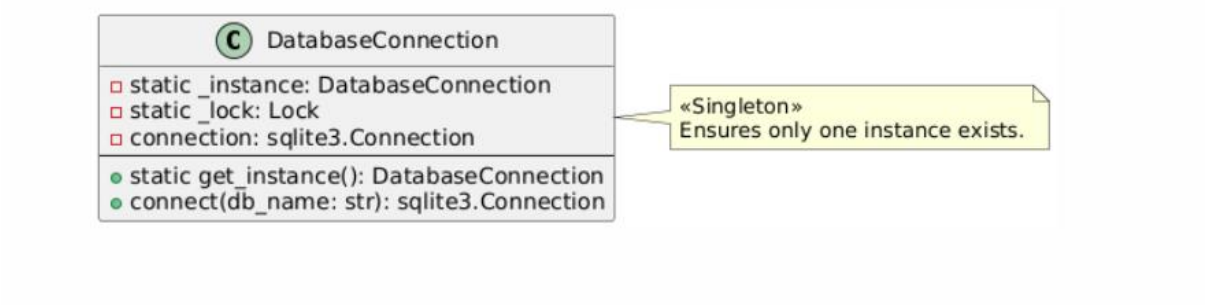
Potential Issues:

1. **Race Conditions:** Multiple threads could simultaneously create separate instances before _instance is fully initialized.
2. **Global State Issues:** If different parts of the application modify the singleton instance, it can lead to **unexpected behavior**.
3. **Testing Challenges:** Singleton makes unit testing harder because it **introduces global state** that persists across tests.

Solutions:

- **Double-Checked Locking**: Ensures only one instance is created even when accessed from multiple threads.
- **Dependency Injection (DI)**: Instead of a strict singleton, pass the database connection as a dependency.
- **Using a Thread-Safe Library**: For databases, use a **connection pool** like SQLAlchemy instead of enforcing a Singleton.

5. Represent using class diagram



---

2. Factory Pattern – Notification System

Concept: Provides an interface for creating different types of objects dynamically.

Questions:

1. How does the Factory pattern improve code maintainability and flexibility?

   The **Factory Pattern** provides a structured way to **create objects dynamically** without modifying the existing codebase. Here's how it improves maintainability and flexibility:

Encapsulation of Object Creation

- The **Factory class** centralizes object creation logic, so changes don't affect the client code.

Easy Expansion

- New notification types (**WhatsApp, Slack, etc.**) can be added without modifying existing code.

Promotes Loose Coupling

- The client **only depends on the interface** (not specific implementations), making the code **more modular**.

Follows Open/Closed Principle

- New notification types can be added **without modifying** existing logic.

2. Implement a Notification interface with a sendNotification() method. Create EmailNotification, SMSNotification, and PushNotification classes that implement this interface.

```
from abc import ABC, abstractmethod


# Step 1: Define an Interface
class Notification(ABC):
    @abstractmethod
    def send_notification(self, message: str):
        pass


# Step 2: Implement Concrete Notification Types
class EmailNotification(Notification):
    def send_notification(self, message: str):
        print(f"Sending Email: {message}")


class SMSNotification(Notification):
    def send_notification(self, message: str):
```

```
        print(f" Sending SMS: {message}")


    class PushNotification(Notification):
        def send_notification(self, message: str):
            print(f" Sending Push Notification: {message}")
```

3. Design a NotificationFactory that returns the correct notification object based on user input (e.g., "email", "sms", or "push").

```
class NotificationFactory:
  @staticmethod
  def create_notification(notification_type: str) -> Notification:
    notification_type = notification_type.lower()  # Normalize input
    if notification_type == "email":
      return EmailNotification()
    elif notification_type == "sms":
      return SMSNotification()
    elif notification_type == "push":
      return PushNotification()
    else:
      raise ValueError("Invalid notification type")
```

Usage Example:

```
factory = NotificationFactory()
notif = factory.create_notification("email")
notif.send_notification("Hello, this is an email notification!")
```

Output:

```
 Sending Email: Hello, this is an email notification!
```

4. Modify the factory to include an additional method for sending a batch of notifications of different types.

```
class NotificationFactory:
    @staticmethod
    def create_notification(notification_type: str) -> Notification:
        notification_type = notification_type.lower()
        if notification_type == "email":
            return EmailNotification()
        elif notification_type == "sms":
            return SMSNotification()
        elif notification_type == "push":
            return PushNotification()
        else:
            raise ValueError("Invalid notification type")

    @staticmethod
    def send_batch_notifications(notifications: list[tuple[str, str]]):
        for notification_type, message in notifications:
            try:
                notification =
NotificationFactory.create_notification(notification_type)
                notification.send_notification(message)
            except ValueError as e:
                print(f"Error: {e}")
Usage Example:
batch = [
    ("email", "Your OTP is 1234"),
```

```
    ("sms", "Your package has been shipped"),
    ("push", "You have a new follower!"),
]

NotificationFactory.send_batch_notifications(batch)
```

Output:

```
Sending Email: Your OTP is 1234
Sending SMS: Your package has been shipped
Sending Push Notification: You have a new follower!
```

5. Represent using class diagram