**import openpyxl**

**from datetime import datetime**

**# --- Classes --- Create------**

**class Wallet:**

**def \_\_init\_\_(self, balance=0.0):**

**self.balance = balance**

**def deposit(self, amount):**

**self.balance += amount**

**def withdraw(self, amount):**

**if self.balance >= amount:**

**self.balance -= amount**

**return True**

**return False**

**def check\_balance(self):**

**return self.balance**

**class User:**

**def \_\_init\_\_(self, user\_id, name, phone\_number, wallet=Wallet()):  #  wallet=Wallet()   --->  Object creation Wallet class**

**self.user\_id = user\_id**

**self.name = name**

**self.phone\_number = phone\_number**

**self.wallet = wallet**

**def display\_details(self):**

**return f"User ID: {self.user\_id}, Name: {self.name}, Phone: {self.phone\_number}"**

**def receive\_money(self, amount):**

**self.wallet.deposit(amount)   #  money recieved and added to wallet**

**class Transaction:**

**def \_\_init\_\_(self, transaction\_id, sender, receiver, amount, date=None):   #  date initial value None**

**self.transaction\_id = transaction\_id**

**self.sender = sender**

**self.receiver = receiver**

**self.amount = amount**

**self.date = f"{datetime.now():%Y-%m-%d %H:%M:%S}"**

**def to\_dict(self):**

**dict = {**

**"Transaction ID": self.transaction\_id,**

**"Sender": self.sender.user\_id,**

**"Receiver": self.receiver.user\_id,**

**"Amount": self.amount,**

**"Date": self.date,**

**}**

**return dict**

**# --- Excel Create & Save Functions ---**

**def save\_users(self, filename="users.xlsx"):**

**wb = openpyxl.Workbook()   #  Create a new workbook**

**sheet = wb.active      #  Get the active worksheet**

**sheet.append(["User ID", "Name", "Phone Number", "Balance"])       #  Add headers to the sheet**

**for user in self.values():   #  Iterate over the values dictionary  ( built-in dictionary method values() )**

**sheet.append([user.user\_id, user.name, user.phone\_number, user.wallet.check\_balance()])**

**wb.save(filename)**

**def save\_transactions(transactions, filename="transactions.xlsx"):**

**# Load the existing file or create a new one**

**try:**

**wb = openpyxl.load\_workbook(filename)**

**except FileNotFoundError:**

**wb = openpyxl.Workbook()**

**sheet = wb.active**

**sheet.append(["Transaction ID", "Sender ID", "Receiver ID", "Amount", "Date"])**

**wb.save(filename)**

**wb = openpyxl.load\_workbook(filename)**

**sheet = wb.active**

**# Get the existing transaction IDs to avoid duplicates**

**existing\_ids = {row[0] for row in sheet.iter\_rows(min\_row=2, values\_only=True)}     #set comprehension**

**# Get the last transaction ID and increment it for new transactions**

**last\_transaction\_id = 0**

**for row in sheet.iter\_rows(min\_row=2, values\_only=True):**

**transaction\_id = row[0]**

**if transaction\_id.startswith('T'):**

**try:**

**last\_transaction\_id = max(last\_transaction\_id, int(transaction\_id[1:]))**

**except ValueError:**

**pass**

**# Append new transactions with auto-incremented transaction IDs**

**for t in transactions:**

**last\_transaction\_id += 1**

**new\_transaction\_id = f"T{last\_transaction\_id:03d}"**

**if new\_transaction\_id not in existing\_ids:**

**sheet.append([new\_transaction\_id, t.sender.user\_id, t.receiver.user\_id, t.amount, t.date])**

**existing\_ids.add(new\_transaction\_id)**

**# Save the updated file**

**wb.save(filename)**

**# --- Main Program ---**

**def main():**

**users = {}          # Initialize an empty dictionary to store users class objects**

**transactions = []   # Initialize an empty list to store transactions class objects**

**print("\nWelcome to the Mobile Payment System!")**

**while True:**

**print("1. Register New User")**

**print("2. Check Balance")**

**print("3. Send Money")**

**print("4. Receive Money")**

**print("5. View Transaction History")**

**print("6. Exit")**

**choice = input("Choose an option: ")**

**if choice == "1":**

**user\_id = input("Enter User ID: ")**

**name = input("Enter Name: ")**

**phone = input("Enter Phone Number: ")**

**if user\_id in users:**

**print("User ID already exists.")**

**else:**

**users[user\_id] = User(user\_id, name, phone)**

**save\_users(users)**

**print("User registered successfully!")**

**elif choice == "2":**

**user\_id = input("Enter User ID: ")**

**if user\_id in users:**

**balance = users[user\_id].wallet.check\_balance()**

**print(f"Current Balance: ${balance:.2f}")**

**else:**

**print("User not found.")**

**elif choice == "3":**

**sender\_id = input("Enter Sender User ID: ")**

**receiver\_id = input("Enter Receiver User ID: ")**

**amount = float(input("Enter Amount: "))**

**if sender\_id in users and receiver\_id in users:**

**sender = users[sender\_id]            # Object of class User**

**receiver = users[receiver\_id]        # Object of class User**

**if sender.wallet.withdraw(amount):   # Use the withdraw method of the wallet class**

**receiver.receive\_money(amount)   # Use the receive\_money method of the User class**

**transaction\_id = f"T{len(transactions)+1:03d}"**

**transaction = Transaction(transaction\_id, sender, receiver, amount)**

**transactions.append(transaction)**

**#save\_users(users)**

**save\_transactions(transactions)   # Save updated balances**

**print(f"Transaction successful! {amount:.2f} sent to {receiver.name}.")**

**else:**

**print("Insufficient balance. Transaction failed.")**

**else:**

**print("Sender or Receiver not found.")**

**elif choice == "4":**

**user\_id = input("Enter User ID : ")**

**receiver\_id = input("Enter Receiver User ID : ")**

**amount = float(input("Enter Amount: "))**

**if user\_id in users and receiver\_id in users:**

**users[user\_id].receive\_money(amount)**

**save\_users(users)  # Save updated balance**

**print(f"Transaction successful! ${amount:.2f} received.")**

**else:**

**print("User not found.")**

**elif choice == "5":**

**print("Transaction History:")**

**for t in transactions:**

**print(f"{t.sender.user\_id} sent ${t.amount:.2f} to {t.receiver.user\_id} on {t.date}")**

**elif choice == "6":**

**#save\_users(users)**

**#save\_transactions(transactions)**

**print("Exiting... Goodbye!")**

**break**

**else:**

**print("Invalid option. Please try again.")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**Which oop concept used in this code and where**

In the code provided, several **Object-Oriented Programming (OOP)** concepts are used. Here's a breakdown of the concepts and where they are applied:

**1. Classes and Objects:**

* **Classes**: Wallet, User, and Transaction are **classes**. A class serves as a blueprint for creating objects (instances).
* **Objects**: When you create instances like sender, receiver, or my\_wallet, they are **objects** of the respective classes.
* **Where**:
  + Wallet: Represents the wallet associated with a user.
  + User: Represents a user who can perform various actions like sending or receiving money.
  + Transaction: Represents a financial transaction between two users.

**2. Encapsulation:**

* **Encapsulation** is the concept of bundling the data (attributes) and the methods (functions) that operate on the data into a single unit or class.
* **Where**:
  + The Wallet class encapsulates the balance and operations such as deposit, withdraw, and check\_balance.
  + The User class encapsulates user information (user\_id, name, phone\_number, and wallet) and actions like receive\_money and display\_details.
  + The Transaction class encapsulates details of a transaction such as transaction\_id, sender, receiver, amount, and date.
* The methods inside the classes provide controlled access to these attributes.

**3. Abstraction:**

* **Abstraction** is the concept of hiding the complex implementation details and showing only the necessary functionality.
* **Where**:
  + The classes Wallet, User, and Transaction abstract away the complexity of how transactions or money management are handled.
  + Users interact with simple methods like withdraw, deposit, and receive\_money without needing to know how these actions are implemented.

**4. Inheritance (though not directly used, can be added):**

* Inheritance allows one class to inherit the attributes and methods of another class.
* **Where**:
  + Although the provided code doesn't directly use inheritance, we could introduce a base class (e.g., Account) and let both User and Wallet inherit from it. This would allow common behavior to be shared.

**5. Polymorphism (through method overriding or method overloading):**

* **Polymorphism** allows methods to do different things based on the object it is acting on. In this case, polymorphism is not directly visible in the provided code but could be introduced.
* **Where**:
  + If we had different types of users (e.g., PremiumUser and BasicUser), we could override methods like send\_money in each subclass to provide different implementations.

**6. Constructor (Initialization):**

* The **constructor** method \_\_init\_\_ is used to initialize the object's attributes when a new object is created.
* **Where**:
  + In the Wallet class, \_\_init\_\_(self, balance=0.0) initializes the balance attribute.
  + In the User class, \_\_init\_\_(self, user\_id, name, phone\_number, wallet=Wallet()) initializes the user\_id, name, phone\_number, and wallet attributes.
  + In the Transaction class, \_\_init\_\_(self, transaction\_id, sender, receiver, amount, date=None) initializes the transaction\_id, sender, receiver, amount, and date attributes.

**7. Methods:**

* Methods are functions defined within a class that operate on its instances (objects).
* **Where**:
  + In Wallet, methods like deposit, withdraw, and check\_balance operate on the wallet's balance.
  + In User, methods like display\_details, receive\_money, and send\_money perform actions on the user's data and wallet.
  + In Transaction, methods like to\_dict help convert the transaction object into a dictionary for saving or further processing.

**8. Composition:**

* **Composition** is when one class contains objects of another class.
* **Where**:
  + The User class **composes** a Wallet object. The user has a wallet (self.wallet), which is an instance of the Wallet class.
  + Similarly, the Transaction class **composes** User objects as its sender and receiver, making it a composition of users.

**Summary of OOP concepts used:**

* **Classes and Objects**: Wallet, User, Transaction.
* **Encapsulation**: All data (e.g., balance, user details) and methods (e.g., deposit, withdraw, receive\_money) are encapsulated within their respective classes.
* **Abstraction**: Complex processes like money transfer and transaction handling are abstracted inside class methods.
* **Constructor (\_\_init\_\_)**: Used to initialize class attributes when creating new objects.
* **Composition**: The User class contains a Wallet object, and the Transaction class contains User objects.

These are the core OOP concepts applied in this code.

**Association**

In the context of **Object-Oriented Programming (OOP)**, the term **associated** generally refers to the relationship between two or more objects, where one object "has" or "contains" another object or the objects are linked to each other. This association is a type of relationship that can be used to model real-world interactions.

**Types of Association in OOP:**

1. **Composition**:
   * **Definition**: A strong association where one object is part of another object. In composition, the contained object cannot exist independently of the containing object. If the container object is destroyed, the contained objects are also destroyed.
   * **Example**: A User class that contains a Wallet object. The User is "associated" with a Wallet, and if the User is deleted, the Wallet will also be deleted.
   * **Where** in the code:

python

Copy code

class User:

def \_\_init\_\_(self, user\_id, name, phone\_number, wallet=Wallet()):

self.user\_id = user\_id

self.name = name

self.phone\_number = phone\_number

self.wallet = wallet # Wallet is associated with User

1. **Aggregation**:
   * **Definition**: A weaker form of association where one object can contain other objects, but the contained objects can exist independently of the container object. In aggregation, the contained objects can be shared between multiple container objects.
   * **Example**: A Library class that contains Book objects. The Book objects can exist independently of the Library.
   * **Where** in the code: You can create multiple Library objects, each containing different Book objects, but Book objects may exist without a Library.
2. **Association** (General Association):
   * **Definition**: A general term for any relationship between two objects where one object uses or interacts with another object. It can be one-way or bi-directional.
   * **Example**: A Transaction object associated with User objects. One user can send money to another user, forming an association between the two.
   * **Where** in the code:

python

Copy code

class Transaction:

def \_\_init\_\_(self, transaction\_id, sender, receiver, amount, date=None):

self.transaction\_id = transaction\_id

self.sender = sender # Sender is associated with the transaction

self.receiver = receiver # Receiver is associated with the transaction

self.amount = amount

self.date = date

**Example of Association in the Provided Code:**

In the code you provided, the **association** occurs in the following places:

* The User object is associated with the Wallet object (composition).
  + **User has a Wallet**. Each user is associated with exactly one wallet, which is initialized when creating the user.
* The Transaction object is associated with two User objects (sender and receiver).
  + **Transaction involves two users**. One user sends money to another, creating an association between the sender and receiver.

**Summary:**

In OOP, **association** refers to the relationship or connection between objects. It can be:

* **Composition**: A strong association where one object is part of another.
* **Aggregation**: A weaker association where objects can exist independently.
* **General Association**: A more general connection where one object interacts or uses another.

In your code, **composition** is the main form of association between the User and Wallet classes, and **general association** is seen in how Transaction associates with User objects as sender and receiver.