1. Sketch the class hierarchy and relationships between all of the classes in this program. Note clearly which classes use each other and any extending/ implementing relationships that exist.

The class hierarchy is structured

- GeneralPlayer (abstract) -> CardPlayer (concrete) - Table> (interface) -> CardTable (concrete)

- CardSet (manages collections of Card objects) - Deck (contains a shuffled collection of Card objects) - Dealer (distributes the deck among players) - Main (executes the game loop) CardPlayer extends GeneralPlayer, giving it a name and defining card-specific behaviors. CardTable implements Table, managing the card placement and player turns. Dealer distributes Card objects from Deck to CardPlayer. The Main class orchestrates the game flow.

The class hierarchy is structured as follows:

- **GeneralPlayer<T>** (abstract) → **CardPlayer** (concrete)
- **Table<T, E extends GeneralPlayer<T>>** (interface) → **CardTable** (concrete)
- **CardSet** (manages collections of `Card` objects)
- **Deck** (contains a shuffled collection of `Card` objects)
- **Dealer** (distributes the deck among players)
- **Main** (executes the game loop)

Relationships:

- `CardPlayer` extends `GeneralPlayer<Card>`, giving it a name and defining card-specific behaviors.
- `CardTable` implements `Table<Card, CardPlayer>`, managing the card placement and player turns.
- `Dealer` distributes `Card` objects from `Deck` to `CardPlayer`.
- `Main` orchestrates the game flow by creating the deck, players, dealer, and table, and running the game loop.

2. Why does it make sense for bank to be a public variable? Does this violate encapsulation rules we have discussed?

Why is bank a Public Variable? bank is public so that other game entities (like CardTable and Main) can directly access and display the player's won cards. This does slightly violate encapsulation but simplifies access since other classes frequently need to check or modify the bank contents. Encapsulation could be preserved by making bank private and providing a getBank() method that returns a copy or read-only view of the bank. However, given the small scope of this game, direct access does not cause major issues.

The `bank` is declared as `public` to allow other game entities (like `CardTable` and `Main`) to directly access and display the player's won cards. This simplifies access since other classes frequently need to check or modify the bank contents.
While this slightly violates encapsulation (as direct access to internal state is generally discouraged), it is acceptable in this context due to the small scope of the

game. A more encapsulated approach would involve making `bank` private and providing a `getBank()` method that returns a copy or read-only view of the bank. However, the current design prioritizes simplicity over strict encapsulation.

3. Describe how you could use the given code to implement a different card game (e.g. poker, blackjack, go fish, etc.). What changes (if any) to the given code would be necessary?

Modifying the Code for a Different Card Game To implement another card game like Poker or Blackjack: - Modify CardPlayer to handle new rules for card selection and scoring. - Modify CardTable to manage different game mechanics (e.g., handling betting in Poker). - Adjust Main to run the new game logic. - Possibly introduce new methods in Card (e.g., face card handling in Blackjack). The existing class structure is general enough to allow these modifications with minimal changes. To implement another card game like Poker or Blackjack:

- **Modify `CardPlayer`**: Add new rules for card selection, scoring, and gameplay (e.g., handling betting in Poker or Ace=1/11 in Blackjack).
- **Modify `CardTable`**: Adjust the table logic to accommodate new game mechanics (e.g., dealing multiple cards, handling community cards in Poker).
- **Adjust `Main`**: Update the game loop to reflect the new rules and scoring system.
- **Introduce New Methods**: For example, add methods in `Card` to handle face cards or special ranks (e.g., Ace in Blackjack).

The existing class structure is general enough to allow these modifications with minimal changes.

4. Describe how you could use the given code to implement a game that does not use playing cards (e.g. chess, tic-tac-toe, checkers, etc.). What changes (if any) to the given code would be necessary?

Modifying the Code for a Non-Card Game To adapt the code for a non-card-based game like Chess or Checkers: - Change Card to a GamePiece class with properties relevant to the new game. - Modify CardSet to handle different piece types. - Adjust CardTable to a GameBoard that tracks piece positions. - Replace CardPlayer with a class that makes moves based on game rules. By modifying class names and adjusting method logic, the current structure could still be reused.

- **Change `Card` to `GamePiece`**: Replace the `Card` class with a `GamePiece` class that represents the pieces used in the new game (e.g., pawns, kings, or X/O markers).
- **Modify `CardSet` to `PieceCollection`**: Update the `CardSet` class to handle different piece types and their properties.
- **Adjust `CardTable` to `GameBoard`**: Replace the table logic with a game board that tracks piece positions and enforces game rules.

- **Replace `CardPlayer`**: Create a new player class that makes moves based on the rules of the new game.

5. Considering your answers to the previous questions, what OOP concepts are important in the ability to reuse this code to implement a different game? Be specific
Key OOP Concepts for Code Reuse - Encapsulation: Separates concerns like card storage (CardSet), player behavior (CardPlayer), and game rules (CardTable). - Inheritance: CardPlayer extends GeneralPlayer, inheriting common functionality while specializing for cards. - Polymorphism: The Table> interface allows different implementations (e.g., adapting CardTable for other games). - Modularity: The design ensures individual components can be changed without affecting others. These principles make it easy to adapt the program to different games with minimal changes.

1. **Encapsulation**: Separates concerns like card storage (`CardSet`), player behavior (`CardPlayer`), and game rules (`CardTable`). This allows individual components to be modified without affecting others.
2. **Inheritance**: `CardPlayer` extends `GeneralPlayer`, inheriting common functionality while specializing for cards. This promotes code reuse and reduces redundancy.
3. **Polymorphism**: The `Table<T, E extends GeneralPlayer<T>>` interface allows different implementations (e.g., adapting `CardTable` for other games). This makes the code flexible and extensible.
4. **Modularity**: The design ensures that individual components (e.g., `Card`, `CardSet`, `Deck`) can be changed or replaced without affecting the overall system.
5. **Abstraction**: The `GeneralPlayer` class abstracts common player behavior, allowing it to be reused for different types of players in various games.