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 with GeneralPlayer<T> as an abstract class, which is extended by the concrete class CardPlayer, creating the possibility of actions for each player in the card game. The interface Table<T, E extends GeneralPlayer<T>> is implemented by CardTable, which manages card placements and each player turns.

To further implement the required rule, the program needs supporting classes, including CardSet, which handles collections of Card objects, and Deck, which contains a shuffled collection of cards. The Dealer distributes the deck among players, ensuring fair card allocation. Finally, Main is the only runnable file, which makes it the class that connects others by creating the deck, players, dealer, and table, running the game, and setting the required format of what is output to the players. These relationships allow the game to function by ensuring proper object interactions, player actions, and necessary management.

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

The bank is declared public so other classes can access and display the player's won cards.

This facilitates easy access because other classes have to check or modify the bank's contents. Although this violates encapsulation to some extent (direct access to the internal state is not recommended), it is acceptable due to the game's low complexity level.

Therefore, 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?

Poker: (The Card class already meets all the requirements for a Poker deck, as it supports ranks and suits)

CardPlayer Changes PokerPlayer:
-Add functionality to handle money, bets, and possibly a bluff system
- Implement logic for card selection (discarding and drawing new cards in games like Five-Card Draw).
- Add methods to evaluate the player's hand (checking for pairs, straights, flushes, etc.)

CardTable Changes:
- Add support for community cards (the flop, turn, and river in Texas Hold 'em).
- Implement betting rounds (pre-flop, flop, turn, river) and manage player interactions during these rounds.
- Handle bluffing by allowing players to fold, call, or raise (this would require input from all players).

Main Changes:
- Update the game loop to reflect Poker's rules (dealing cards, managing betting rounds, and determining the winner based on hand strength).
- Add logic to handle multiple players (if extending beyond two players).

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?

Chess:

Card becomes Piece:
- We have to forget suits and ranks to represent the pieces (pawns, knights, bishops, rooks, queens, and kings). Each has its own style of movement, so we would have to add properties like pieceType and color (white and black). Depending on the complex level, the movements of en passant, castle, and pawn becoming any piece besides the king should also be implemented
CardSet becomes PieceSet:
- Now, this class has to handle different piece types, keeping track of the pieces for each player

CardTable becomes GameBoard:
- Replace the table with a GameBoard class that tracks piece positions on an 8x8 grid
- Implement methods to move pieces, enforce Chess rules (check, double check, checkmate), and update the board state

CardPlayer becomes ChessPlayer:
- New type of player
- Creating unique actions for each player in the card 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

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 (adapting CardTable for other games). This makes the code flexible and extensible.
4. Modularity: The design ensures that individual components (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.