AGH University of Science and Technology

Faculty of Mechanical Engineering and Robotics

Project of the Blackjack by MR I.Rybalko and Mr K.M.Thant

Team members:

- Ivan Rybalko
- KyiMin Thant

Summary:

1. Description:

Blackjack, also known as "21", is one of the most popular card games in the world. The objective is for each player to draw cards and achieve a total score as close to 21 as possible without exceeding it. Players compete against a dealer, not against each other.

In this project, a s version of Blackjack is implemented in C++ using principles of object-oriented programming (OOP). The program runs in a console environment and supports two human players playing against an dealer.

1.1 Architecture of the project:

The architecture of the project is divided into several classes:

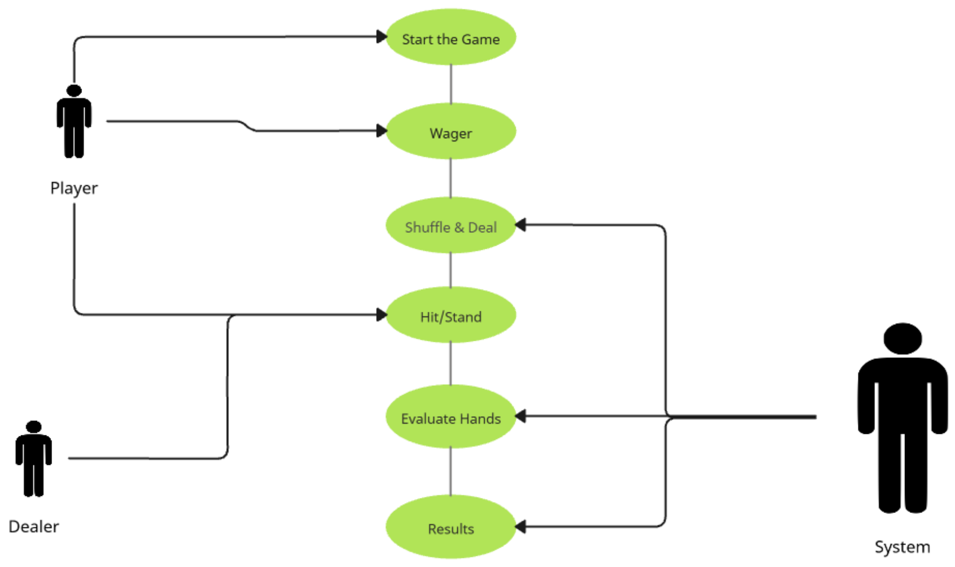
1. Cards: represents a single card with rank and suit.
2. Deck: manages a collection of shuffled cards and deals them.
3. Hand: contains the cards currently held by a player or dealer and calculates the total score.
4. Player: an abstract base class that defines shared functionality for all players.
5. PlayerHuman: derived class that allows user input to perform moves.
6. Dealer: derived class that implements automated rules for the dealer's behavior.
7. System: coordinates the entire game loop, including dealing, betting, handling player turns, and determining outcomes.

A more detailed description of the architecture, including classes, methods, and functional structure, will be provided in Section 3.

2. Use Case Diagram:

The following use case diagram ,Figure 1, represents the main interactions between the system and the actors involved in the Blackjack game: **Players**, **Dealer**, and the central **System** controller.

- **Player** can initiate the game, place a wager, and perform actions like **Hit** , **Stand or double** based on the game state.
- **Dealer** automatically participates in the game by receiving cards and acting according to predefined rules .
- The **System** manages the entire flow: it starts the game, shuffles and deals cards, manages betting and player actions, evaluates the outcome of the round, and displays the results.



2.1 Detailed Architecture Overview

The project is structured according to **object-oriented programming (OOP)** principles. The game logic is separated into multiple classes, each responsible for a specific role within the system.

The structure of the program is built on clearly defined **interactions between classes**, where each component serves a dedicated role within the system.

At the core is the System class, which acts as the **game controller**. It holds and coordinates the following key components:

- A single Deck instance that handles card initialization and shuffling.
- Two PlayerHuman objects (player1 and player2), each representing a human-controlled participant.
- One Dealer object, which operates automatically based on Blackjack rules.

The Player base class (parent of both PlayerHuman and Dealer) contains shared attributes such as the player's name, balance, bet, and an instance of the Hand class, which stores the cards currently held.

The Hand class internally uses a vector<Cards> to store individual Cards, and calculates the score by interpreting card values (e.g., handling Aces as 1 or 11).

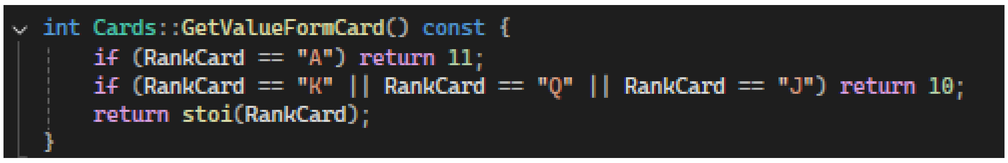
During gameplay:

- The System creates and shuffles the deck via Deck.
- It distributes cards and manages the round via method calls to each PlayerHuman and the Dealer.
- Each player uses their Hand to receive cards and compute scores.
- The System handles betting logic, compares hands, and prints results through displayResults() and playRound().

This architecture ensures **modular separation**, **code reuse** (via inheritance), and **centralized flow control** in a maintainable and scalable way.

Cards

- **Responsibility:** Represents a single playing card.
- **Attributes:**
  - string RankCard: the rank ("2", "J", "A", etc.)
  - char Symbol: the suit ('S', 'H', 'D', 'C')
- **Key Methods:**
  - **GetValueFormCard()**: returns the card's value based on its rank



```cpp
int Cards::GetValueFormCard() const {
    if (RankCard == "A") return 11;
    if (RankCard == "K" || RankCard == "Q" || RankCard == "J") return 10;
    return stoi(RankCard);
}
```

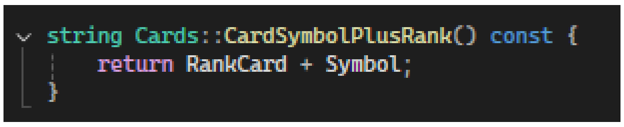
Logic:

- Returns 11 for an Ace ("A").
- Returns 10 for face cards ("K", "Q", "J").
- For numbered cards (2–10), converts the rank to an integer using stoi.

Return: Integer value of the card (e.g., 11 for Ace, 10 for King, 5 for "5").

Usage: Used by the Hand class to calculate the total score of a player's or dealer's hand.

- **CardSymbolPlusRank()**: returns a string for display (e.g. "AS")



```cpp
string Cards::CardSymbolPlusRank() const {
    return RankCard + Symbol;
}
```

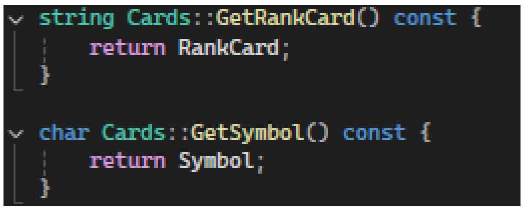
Logic: Concatenates the RankCard and Symbol attributes.

Return: String combining rank and suit (e.g., "10H", "AS").

Usage: Used in console output to show cards in a human-readable format during gameplay.

Example: For a card with RankCard = "A" and Symbol = 'S', returns "AS".

- **GetRankCard()**, **GetSymbol()**: accessors



```cpp
string Cards::GetRankCard() const {
    return RankCard;
}

char Cards::GetSymbol() const {
    return Symbol;
}
```

GetRankCard()

Return: String representing the card's rank (e.g., "A", "7").

Usage: Used by other classes (e.g., Hand) to check the rank, particularly for Ace detection in score calculations.

Example: For an Ace, returns "A".

GetSymbol()

Return: Char representing the card's suit ('S', 'H', 'D', 'C').

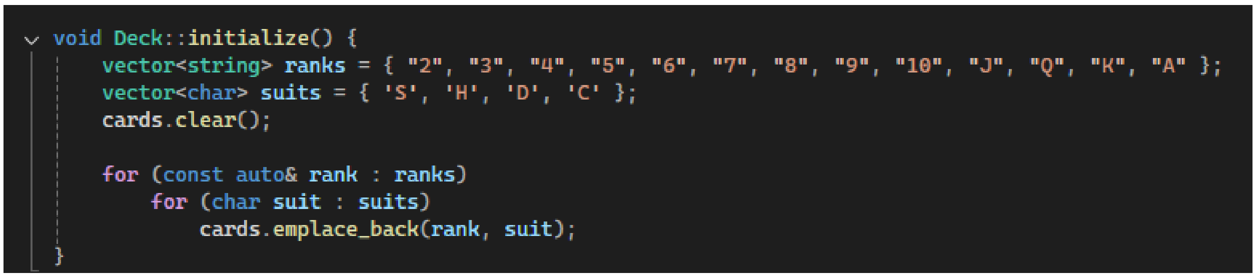
Usage: Used when the suit needs to be accessed separately, though less common in Blackjack logic.

Example: For a card with Symbol = 'H', returns 'H'.

Deck

- **Responsibility:** Manages a full deck of 52 cards.
- **Attributes:**
    - vector<Cards> cards: the current card stack

- **Key Methods:**
    - **initialize()**: creates a new 52-card deck



```
void Deck::initialize() {
    vector<string> ranks = { "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A" };
    vector<char> suits = { 'S', 'H', 'D', 'C' };
    cards.clear();

    for (const auto& rank : ranks)
        for (char suit : suits)
            cards.emplace_back(rank, suit);
}
```

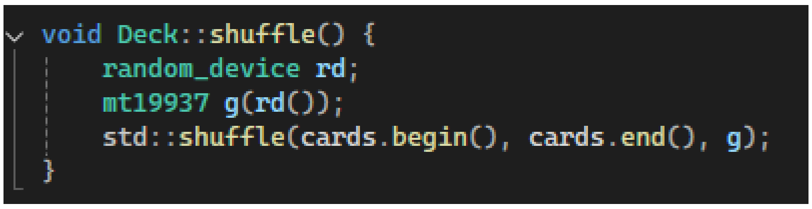
Logic:

- Defines arrays of ranks ("2" to "A") and suits ('S', 'H', 'D', 'C').
- Clears the existing cards vector.
- Iterates through all rank-suit combinations to create Cards objects and adds them to the cards vector.

Usage: Called at the start of each round to reset the deck.

Example: Populates cards with 52 unique Cards objects (e.g., "2S", "AH").

- **shuffle()**: shuffles cards using Mersenne Twister (std::mt19937)



```
void Deck::shuffle() {
    random_device rd;
    mt19937 g(rd());
    std::shuffle(cards.begin(), cards.end(), g);
}
```

Logic:

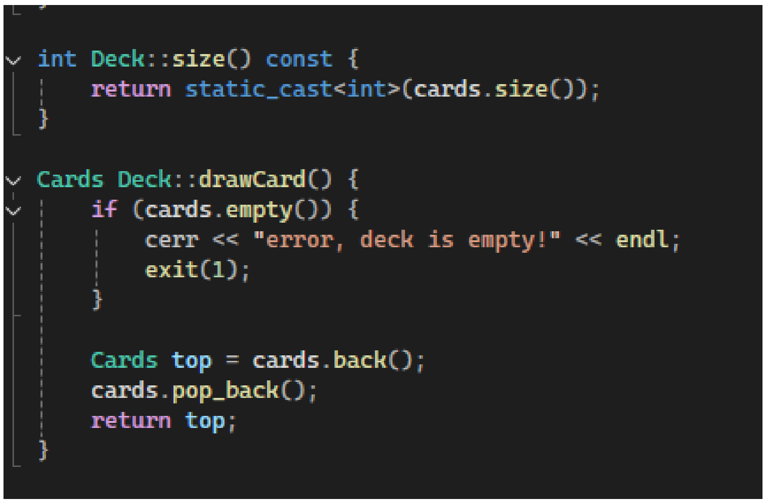
- Uses std::random_device to seed a Mersenne Twister random number generator (std::mt19937).

- Applies std::shuffle to reorder the cards vector.

Usage: Called after initialize() to ensure cards are dealt in random order.

Example: Randomly reorders the 52 cards in the cards vector.

  - **drawCard()**: removes and returns the top card



```cpp
int Deck::size() const {
    return static_cast<int>(cards.size());
}

Cards Deck::drawCard() {
    if (cards.empty()) {
        cerr << "error, deck is empty!" << endl;
        exit(1);
    }

    Cards top = cards.back();
    cards.pop_back();
    return top;
}
```

Logic:

- Checks if the deck is empty; if so, outputs an error message and exits the program.
- Retrieves the last card in the cards vector, removes it using pop_back(), and returns it.

Return: A Cards object representing the drawn card.

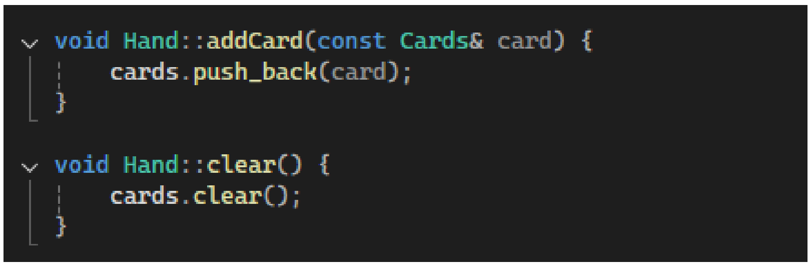
Usage: Called during gameplay to deal cards to players or the dealer.

Error Handling: Terminates the program if the deck is empty.

Example: Draws and returns a card like "7H", reducing the deck size by 1.

Hand

- **Responsibility:** Holds the cards of a player or dealer.
- **Attributes:**
  - vector<Cards> cards: list of cards in hand
- **Key Methods:**
  - **addCard()**, **clear()**: manage cards



```cpp
void Hand::addCard(const Cards& card) {
    cards.push_back(card);
}

void Hand::clear() {
    cards.clear();
}
```

addCard()

Parameters: (const Cards& card): The card to add.

Logic: Appends the card to the cards vector using push_back.

Usage: Called when a player or dealer receives a card via receiveCard.

Example: Adds a card like "AS" to the hand's cards vector.

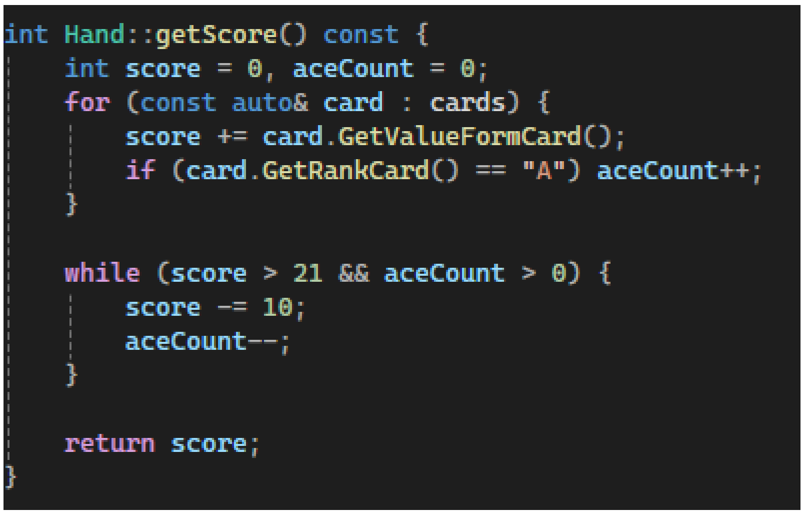
clear()

Logic: Clears the cards vector.

Usage: Called at the start of a new round to reset the player's or dealer's hand.

Example: Empties the cards vector, preparing for a new round.

- **getScore()**: calculate total value, treating Aces as 11 or 1



```
int Hand::getScore() const {
    int score = 0, aceCount = 0;
    for (const auto& card : cards) {
        score += card.GetValueFormCard();
        if (card.GetRankCard() == "A") aceCount++;
    }

    while (score > 21 && aceCount > 0) {
        score -= 10;
        aceCount--;
    }

    return score;
}
```

Logic:

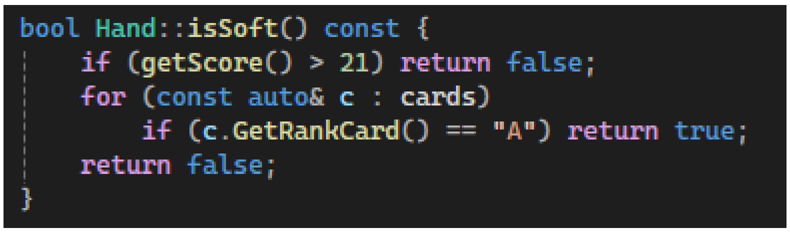
- Iterates through cards, summing the values from GetValueFormCard().
- Counts Aces (initially valued at 11).
- If the score exceeds 21 and Aces are present, reduces the score by 10 per Ace (treating it as 1) until the score is 21 or less or no Aces remain.

Return: Integer score of the hand.

Usage: Used to determine the player's or dealer's score for game outcomes.

Example: For cards ["10H", "AS"], returns 21 (10 + 11) or 11 (10 + 1) if busting.

- **isSoft()**: determines if hand contains a "soft" Ace



```cpp
bool Hand::isSoft() const {
    if (getScore() > 21) return false;
    for (const auto& c : cards)
        if (c.GetRankCard() == "A") return true;
    return false;
}
```

Logic:

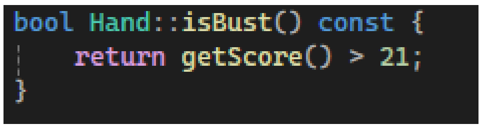
- Returns false if the score exceeds 21 (bust).
- Checks for any card with rank "A".

Return: true if the hand has an Ace and score ≤ 21, false otherwise.

Usage: Used by the Dealer to decide whether to hit on a soft 17.

Example: For ["AS", "6H"] (score 17), returns true (soft 17).

- **isBust()**: checks for score > 21



```cpp
bool Hand::isBust() const {
    return getScore() > 21;
}
```

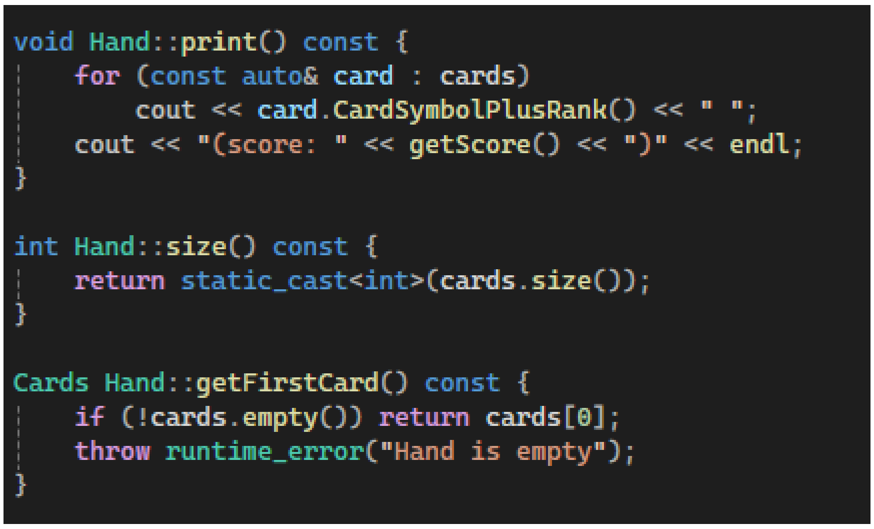
Logic: Calls getScore() and checks if the result is greater than 21.

Return: true if score > 21, false otherwise.

Usage: Used to check if a player or dealer has lost by going over 21.

Example: For ["10H", "QH", "5S"] (score 25), returns true.

- **print(), getFirstCard()**



```cpp
void Hand::print() const {
    for (const auto& card : cards)
        cout << card.CardSymbolPlusRank() << " ";
    cout << "(score: " << getScore() << ")" << endl;
}

int Hand::size() const {
    return static_cast<int>(cards.size());
}

Cards Hand::getFirstCard() const {
    if (!cards.empty()) return cards[0];
    throw runtime_error("Hand is empty");
}
```

print()

Logic:

- Iterates through cards, printing each card’s CardSymbolPlusRank() output.
- Prints the total score using getScore().

Usage: Used during gameplay to show the player’s or dealer’s hand in the console.

Example: For ["AS", "10H"], outputs "AS 10H (score: 21)".

getFirstCard()

Logic:

- Checks if cards is not empty; if empty, throws a runtime_error.
- Returns the first card in the cards vector.

Return: A Cards object representing the first card.

Usage: Used to display the dealer’s visible card to players.

Error Handling: Throws an exception if the hand is empty.

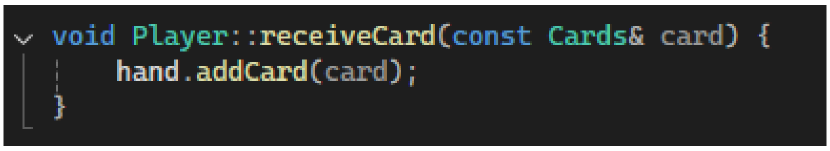
Example: For ["KS", "7H"], returns the Cards object for "KS".

Player

- **Responsibility:** Base class for all player types.
- **Attributes:**
    - name, balance, currentBet, Hand hand
- **Key Methods:**
    - **makeMove()** – virtual

makeMove() is virtual void and we override this in **playerHuman** and **Dealer**(child classes of Player).

- - **receiveCard()**, **placeBet()**, **adjustBalance()**



```
void Player::receiveCard(const Cards& card) {
    hand.addCard(card);
}
```

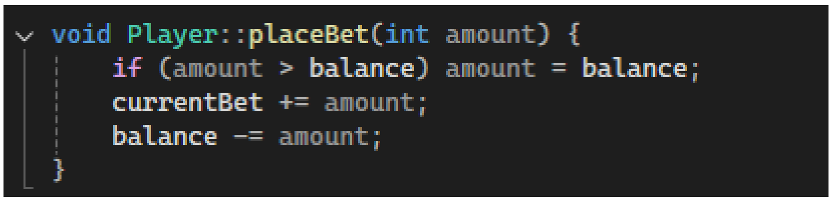
receiveCard()

Parameters: (const Cards& card): The card to add.

Logic: Calls hand.addCard(card) to append the card to the player’s Hand.

Usage: Called during dealing or when a player hits.

Example: Adds a card like "7S" to the player’s hand.



```cpp
void Player::placeBet(int amount) {
    if (amount > balance) amount = balance;
    currentBet += amount;
    balance -= amount;
}
```

placeBet()

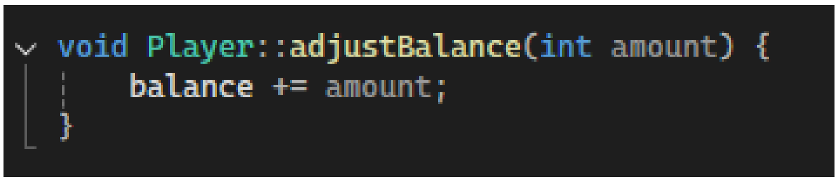
Parameters: amount (int): The bet amount.

Logic:

- If amount exceeds balance, sets amount to balance.
- Adds amount to currentBet and subtracts it from balance.

Usage: Called when players place bets at the start of a round or double down.

Example: For amount = 50 and balance = 100, sets currentBet = 50, balance = 50.



```cpp
void Player::adjustBalance(int amount) {
    balance += amount;
}
```

adjustBalance()

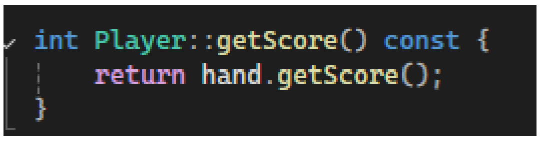
Parameters: amount (int): Amount to add (positive for wins, negative for losses).

Logic: Adds amount to balance.

Usage: Called in displayResults to update balance based on game outcomes.

Example: For amount = 100, increases balance by 100

- ○ **getScore(), isBust(), getVisibleCard(), addMoney()**



```cpp
int Player::getScore() const {
    return hand.getScore();
}
```

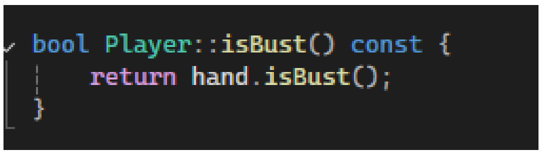
getScore()

Logic: Calls hand.getScore() to get the hand's score.

Return: Integer score of the hand.

Usage: Used to evaluate the player's standing in the game.

Example: Returns 21 for a hand like ["AS", "10H"].



```cpp
bool Player::isBust() const {
    return hand.isBust();
}
```

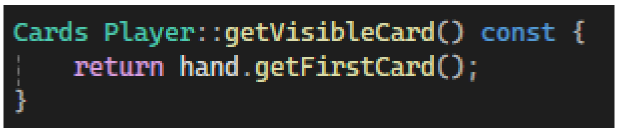
isBust()

Logic: Calls hand.isBust() to check the hand’s status.

Return: true if score > 21, false otherwise.

Usage: Used to determine if the player has lost the round.

Example: Returns true for ["10H", "QH", "5S"] (score 25).



```cpp
Cards Player::getVisibleCard() const {
    return hand.getFirstCard();
}
```

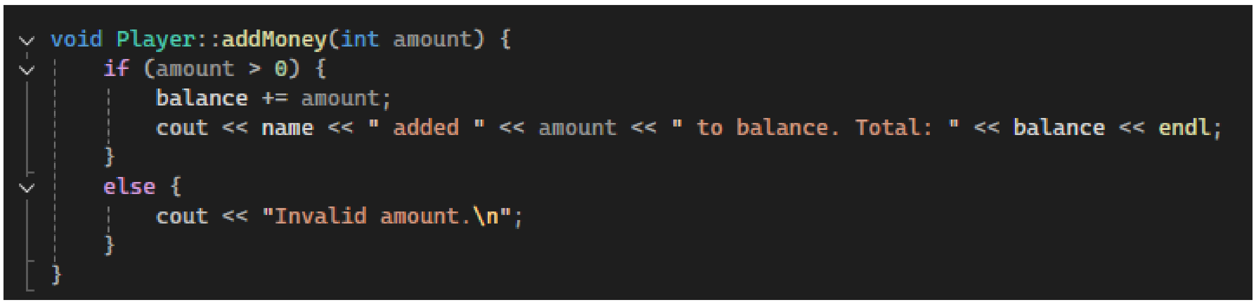
getVisibleCard()

Logic: Calls hand.getFirstCard() to return the first card.

Return: A Cards object representing the first card.

Usage: Used to show the dealer’s visible card to players.

Example: Returns "KS" for a hand starting with "KS".



```cpp
void Player::addMoney(int amount) {
    if (amount > 0) {
        balance += amount;
        cout << name << " added " << amount << " to balance. Total: " << balance << endl;
    }
    else {
        cout << "Invalid amount.\n";
    }
}
```

addMoney()

Parameters: amount (int): The amount to add.

Logic:

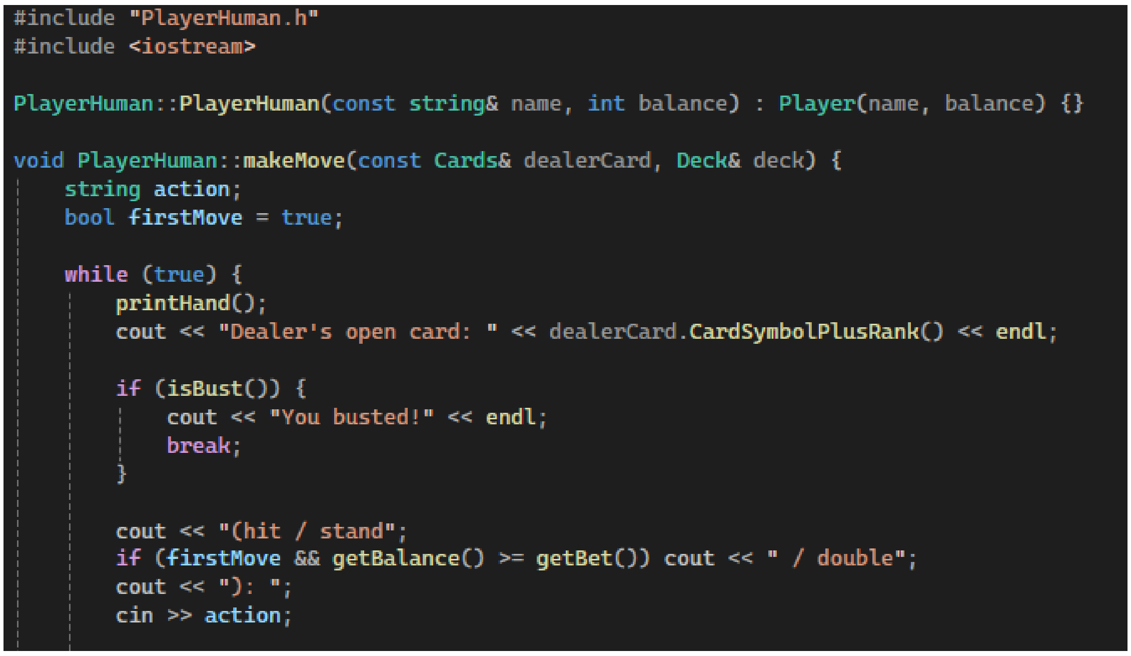
- If amount > 0, adds amount to balance and prints a confirmation.
- If amount <= 0, prints an error message.

Usage: Called by System::handleTopUp to allow players to add funds.

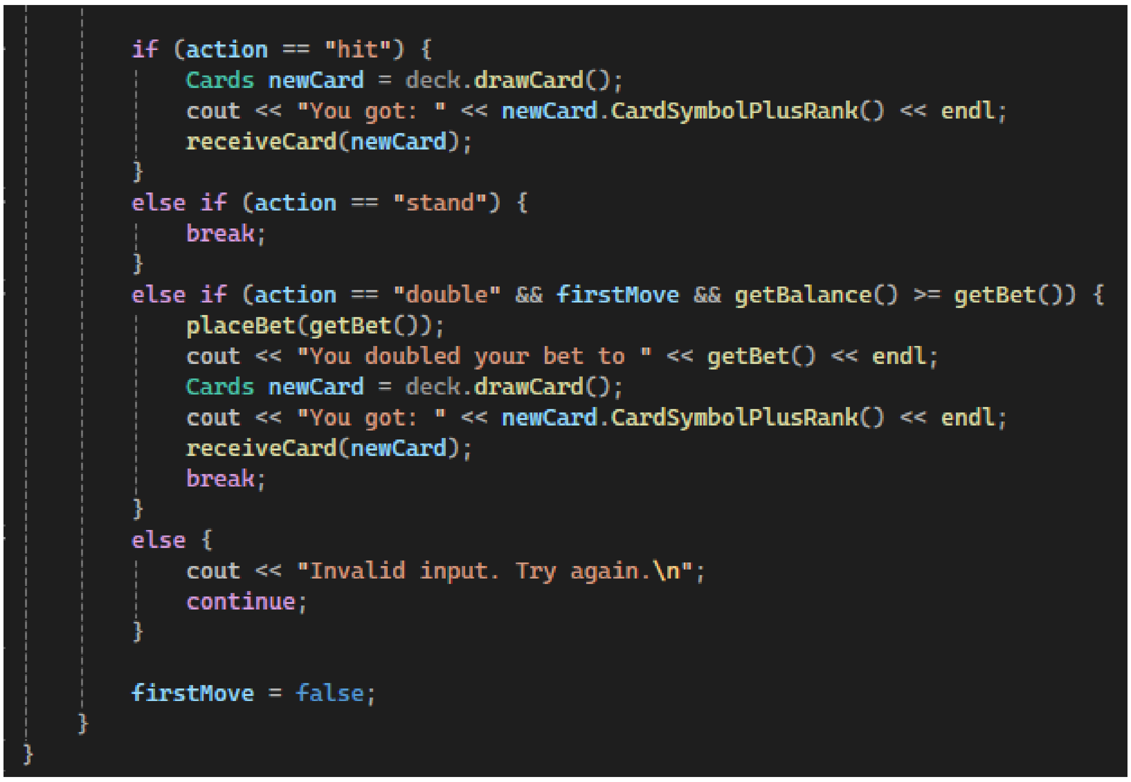
Example: For amount = 100, increases balance by 100 and outputs a message.

PlayerHuman : Player

- **Responsibility:** Implements user input during gameplay.
- **Key Method:**
  - **makeMove()**: handles hit, stand, and double based on user input



```cpp
#include "PlayerHuman.h"
#include <iostream>

PlayerHuman::PlayerHuman(const string& name, int balance) : Player(name, balance) {}

void PlayerHuman::makeMove(const Cards& dealerCard, Deck& deck) {
    string action;
    bool firstMove = true;

    while (true) {
        printHand();
        cout << "Dealer's open card: " << dealerCard.CardSymbolPlusRank() << endl;

        if (isBust()) {
            cout << "You busted!" << endl;
            break;
        }

        cout << "(hit / stand";
        if (firstMove && getBalance() >= getBet()) cout << " / double";
        cout << "): ";
        cin >> action;
```



```cpp
        if (action == "hit") {
            Cards newCard = deck.drawCard();
            cout << "You got: " << newCard.CardSymbolPlusRank() << endl;
            receiveCard(newCard);
        }
        else if (action == "stand") {
            break;
        }
        else if (action == "double" && firstMove && getBalance() >= getBet()) {
            placeBet(getBet());
            cout << "You doubled your bet to " << getBet() << endl;
            Cards newCard = deck.drawCard();
            cout << "You got: " << newCard.CardSymbolPlusRank() << endl;
            receiveCard(newCard);
            break;
        }
        else {
            cout << "Invalid input. Try again.\n";
            continue;
        }

        firstMove = false;
    }
}
```

Purpose: Handles the player’s actions based on user input.

Parameters:

- (const Cards& dealerCard): The dealer’s visible card.
- (Deck& deck): The deck to draw cards from.

Logic:

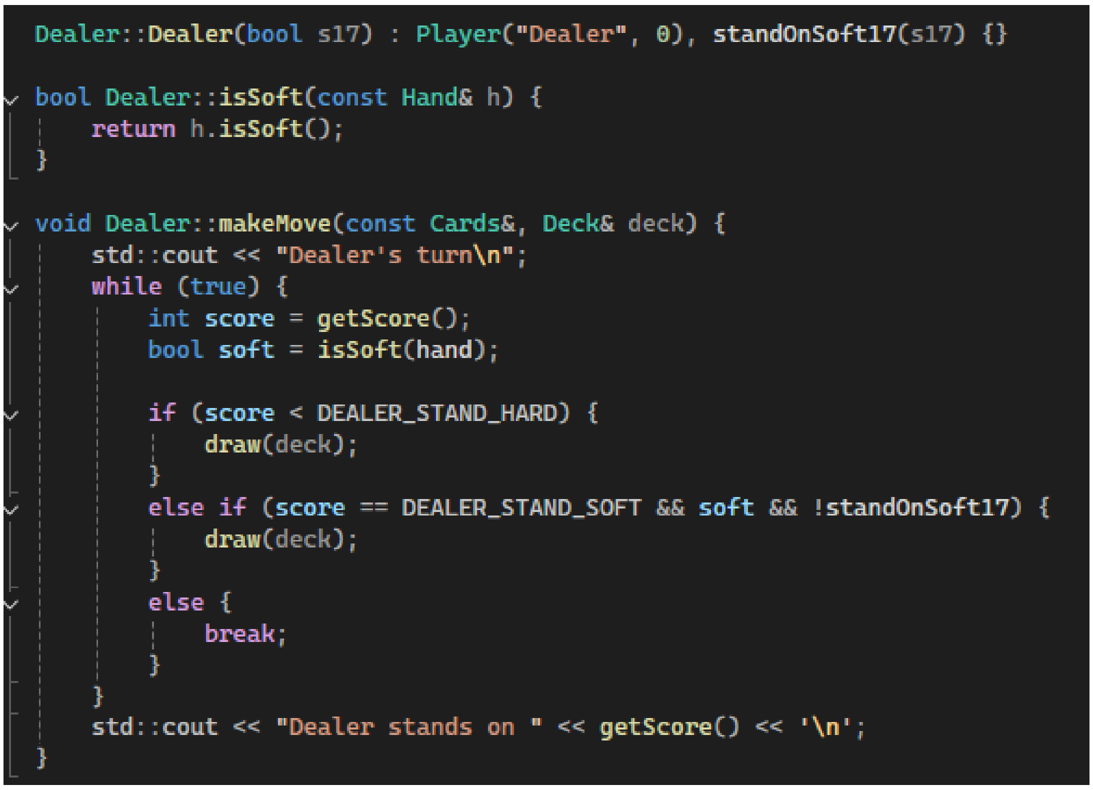
- Enters a loop until the player stands, busts, or doubles down.
- Displays the player's hand and the dealer's visible card.
- Prompts for input ("hit", "stand", or "double" if it's the first move and balance allows).
- For "hit": Draws a card, displays it, and adds it to the hand.
- For "stand": Exits the loop.
- For "double" (if valid): Doubles the bet, draws one card, adds it, and exits.
- For invalid input: Prints an error and continues the loop.
- Sets firstMove = false after the first action to disable doubling.

Usage: Called during the player's turn in System::playRound.

Example: If the user inputs "hit", draws a card like "7S" and updates the hand.

Dealer : Player

- **Responsibility:** Plays automatically following Blackjack rules.
- **Key Method:**
    - **makeMove()**: draws until score reaches 17 (or soft 17, depending on setting)



```cpp
Dealer::Dealer(bool s17) : Player("Dealer", 0), standOnSoft17(s17) {}

bool Dealer::isSoft(const Hand& h) {
    return h.isSoft();
}

void Dealer::makeMove(const Cards&, Deck& deck) {
    std::cout << "Dealer's turn\n";
    while (true) {
        int score = getScore();
        bool soft = isSoft(hand);

        if (score < DEALER_STAND_HARD) {
            draw(deck);
        }
        else if (score == DEALER_STAND_SOFT && soft && !standOnSoft17) {
            draw(deck);
        }
        else {
            break;
        }
    }
    std::cout << "Dealer stands on " << getScore() << '\n';
}
```

Purpose: Automates the dealer's actions based on Blackjack rules.

Parameters:

- Ignored Cards& parameter (included for interface consistency).

- (Deck& deck): The deck to draw cards from.

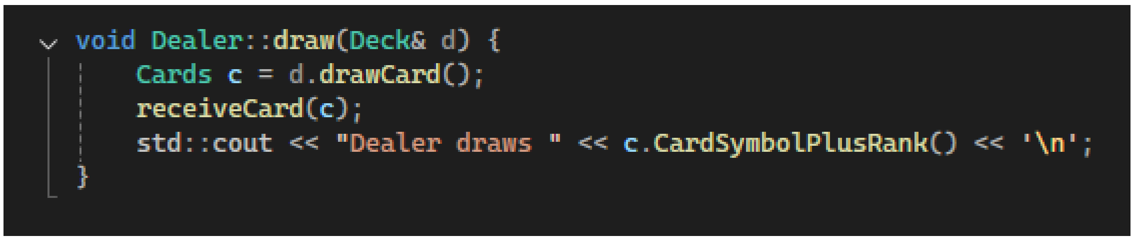
Logic:

- Prints "Dealer's turn".
- Enters a loop:
    - If score < DEALER_STAND_HARD (17), draws a card.
    - If score = DEALER_STAND_SOFT (17) and the hand is soft and standOnSoft17 is false, draws a card.
    - Otherwise, breaks the loop.
- Prints the final score when standing.

Usage: Called in System::playRound after players' turns.

Example: Draws cards until the score is ≥ 17, e.g., stops at ["KS", "7S"] (score 17).

- **draw()**: helper function to draw and display a card



```
void Dealer::draw(Deck& d) {
    Cards c = d.drawCard();
    receiveCard(c);
    std::cout << "Dealer draws " << c.CardSymbolPlusRank() << '\n';
}
```

Parameters: (Deck& d): The deck to draw from.

Logic:

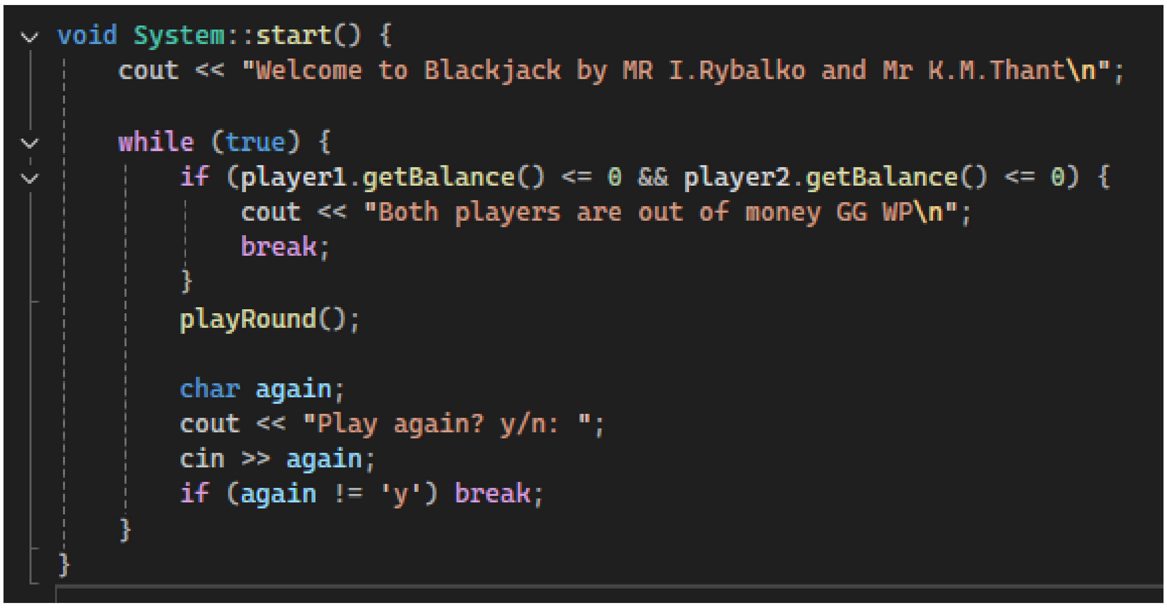
- Draws a card using d.drawCard().
- Adds it to the hand using receiveCard.
- Prints the drawn card's CardSymbolPlusRank.

Usage: Called within makeMove to handle card drawing.

Example: Draws "7S", outputs "Dealer draws 7S", and adds it to the hand.

System

- **Responsibility:** Manages the entire game flow.
- **Key Methods:**
    - start(): main game loop



```cpp
void System::start() {
    cout << "Welcome to Blackjack by MR I.Rybalko and Mr K.M.Thant\n";

    while (true) {
        if (player1.getBalance() <= 0 && player2.getBalance() <= 0) {
            cout << "Both players are out of money GG WP\n";
            break;
        }
        playRound();

        char again;
        cout << "Play again? y/n: ";
        cin >> again;
        if (again != 'y') break;
    }
}
```

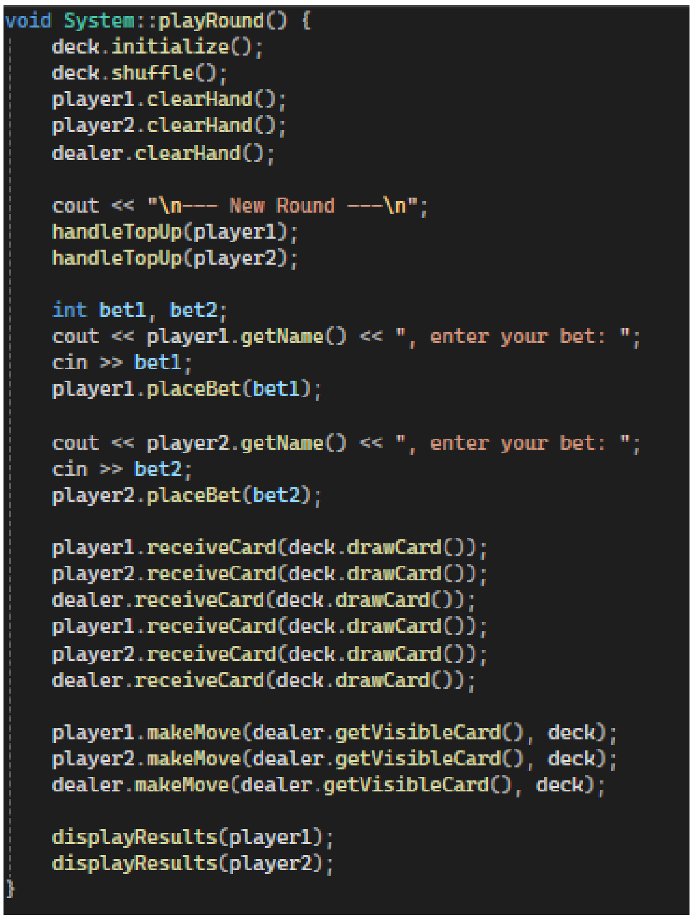
Logic:

- Prints a welcome message.
- Enters a loop that continues until both players have zero balance.
- Calls playRound() for each round.
- Prompts to play again ("y/n"); exits if not "y".

Usage: Entry point for the game, called in main().

Example: Runs multiple rounds until players are out of money or choose to exit.

- **playRound()**: one round of gameplay



```cpp
void System::playRound() {
    deck.initialize();
    deck.shuffle();
    player1.clearHand();
    player2.clearHand();
    dealer.clearHand();

    cout << "\n--- New Round ---\n";
    handleTopUp(player1);
    handleTopUp(player2);

    int bet1, bet2;
    cout << player1.getName() << ", enter your bet: ";
    cin >> bet1;
    player1.placeBet(bet1);

    cout << player2.getName() << ", enter your bet: ";
    cin >> bet2;
    player2.placeBet(bet2);

    player1.receiveCard(deck.drawCard());
    player2.receiveCard(deck.drawCard());
    dealer.receiveCard(deck.drawCard());
    player1.receiveCard(deck.drawCard());
    player2.receiveCard(deck.drawCard());
    dealer.receiveCard(deck.drawCard());

    player1.makeMove(dealer.getVisibleCard(), deck);
    player2.makeMove(dealer.getVisibleCard(), deck);
    dealer.makeMove(dealer.getVisibleCard(), deck);

    displayResults(player1);
    displayResults(player2);
}
```

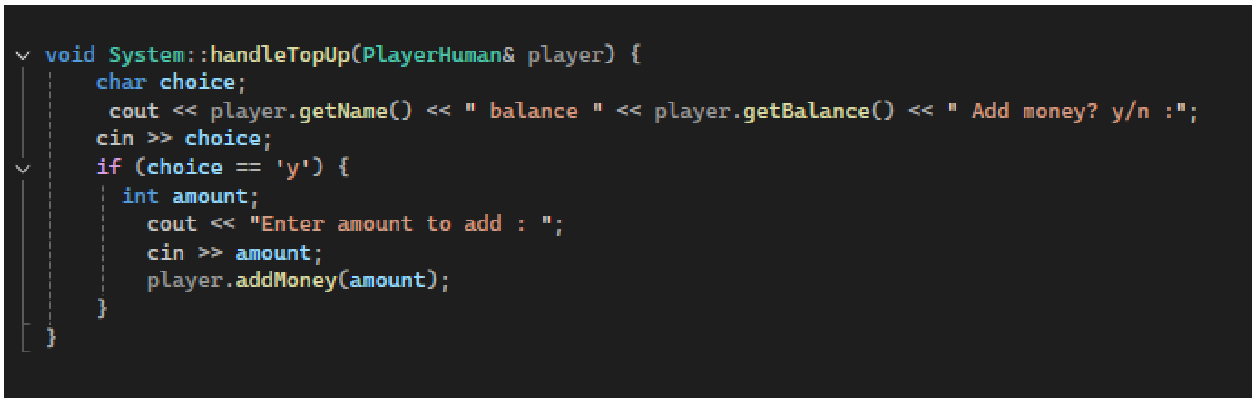
Logic:

- Initializes and shuffles the deck.
- Clears hands for both players and the dealer.
- Calls handleTopUp for each player.
- Prompts each player for a bet and calls placeBet.
- Deals two cards to each player and the dealer.
- Calls makeMove for each player and the dealer.
- Calls displayResults for each player.

Usage: Called in start() to handle one round.

Example: Manages a round where players bet, receive cards, make moves, and see results.

- **handleTopUp()**: allows players to add money



```cpp
void System::handleTopUp(PlayerHuman& player) {
    char choice;
     cout << player.getName() << " balance " << player.getBalance() << " Add money? y/n :";
    cin >> choice;
    if (choice == 'y') {
      int amount;
        cout << "Enter amount to add : ";
        cin >> amount;
        player.addMoney(amount);
    }
}
```

Parameters: (PlayerHuman& player): The player to top up.

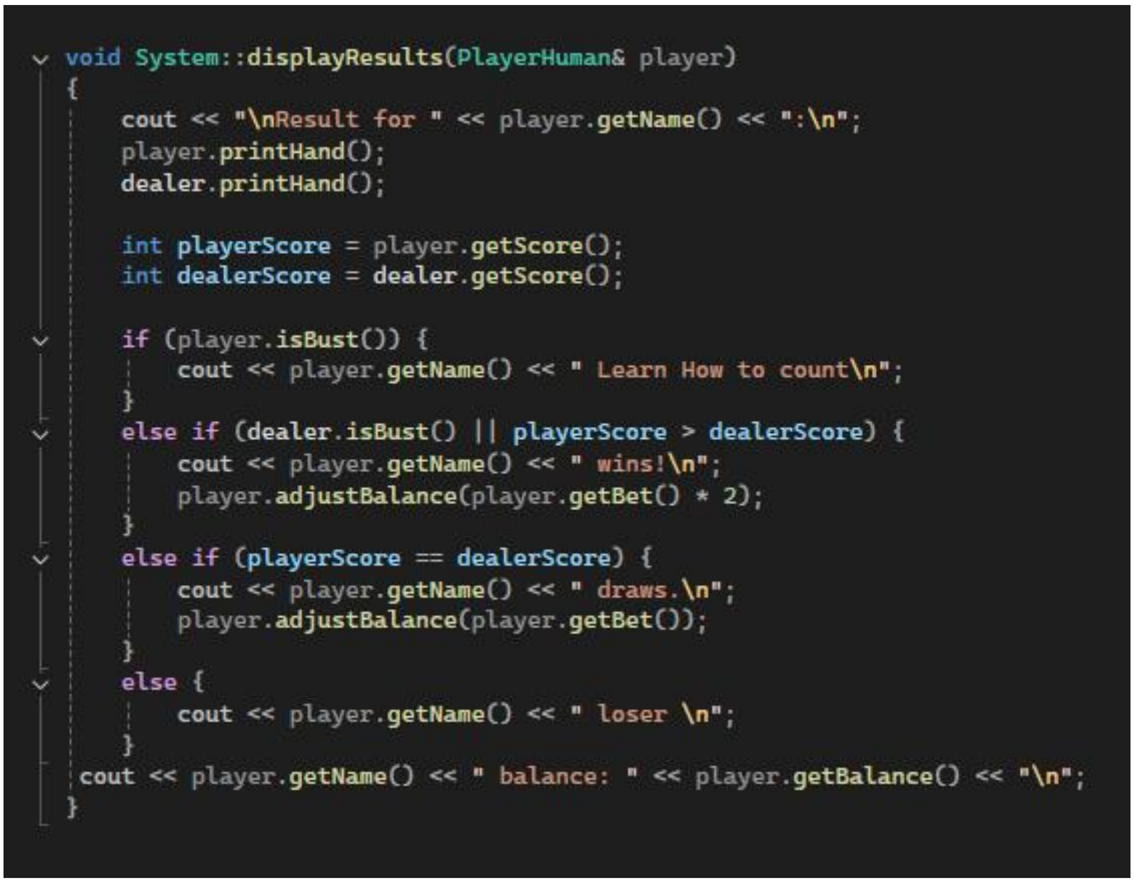
Logic:

- Displays the player's current balance.
- Prompts for a yes/no choice to add money.
- If "y", prompts for an amount and calls player.addMoney.

Usage: Called in playRound before betting.

Example: For "Ryba" with balance 100, adds 50 if the user inputs "y" and 50.

- **displayResults()**: compares scores and prints outcome



```cpp
void System::displayResults(PlayerHuman& player)
{
    cout << "\nResult for " << player.getName() << ":\n";
    player.printHand();
    dealer.printHand();

    int playerScore = player.getScore();
    int dealerScore = dealer.getScore();

    if (player.isBust()) {
        cout << player.getName() << " Learn How to count\n";
    }
    else if (dealer.isBust() || playerScore > dealerScore) {
        cout << player.getName() << " wins!\n";
        player.adjustBalance(player.getBet() * 2);
    }
    else if (playerScore == dealerScore) {
        cout << player.getName() << " draws.\n";
        player.adjustBalance(player.getBet());
    }
    else {
        cout << player.getName() << " loser \n";
    }
 cout << player.getName() << " balance: " << player.getBalance() << "\n";
}
```

Parameters: (PlayerHuman& player): The player to evaluate.

Logic:

- Prints the player’s and dealer’s hands.
- Compares scores:
    - If player busts, outputs a loss message.
    - If dealer busts or player’s score > dealer’s, player wins (receives 2x bet).
    - If scores are equal, player draws (receives bet back).
    - Otherwise, player loses.
- Updates balance using adjustBalance.
- Prints the updated balance.

Usage: Called in playRound to show results for each player.

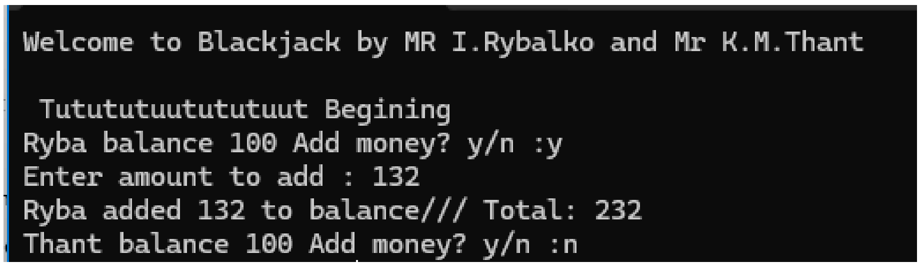
Example: For Ryba with score 22 and dealer with 17, outputs "Ryba busted" and updates balance.

3. Program Flow and Execution Logic

The Blackjack program follows a structured, step-by-step execution flow controlled by the System class. Below is a detailed description of how the program behaves from start to finish, including user interaction and console output.

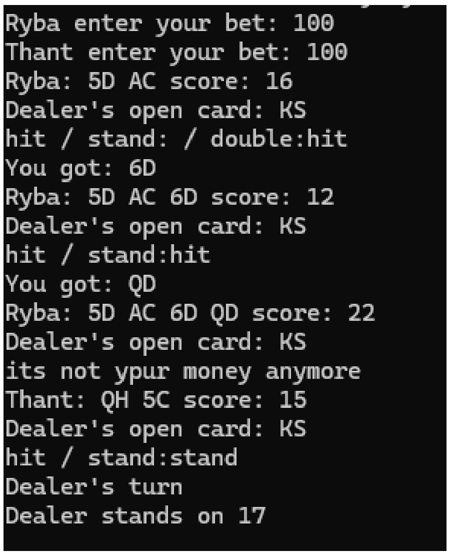
1. **Program starts in main()**:
   - A System object is created.
   - The start() method is called.
2. **Inside System::start()**:
   - A welcome message is printed.
   - The game enters a while loop that continues as long as at least one player has money.
3. **Each iteration of the loop = one game round (playRound())**:
   - A new Deck is initialized and shuffled.
   - Hands for both players and the dealer are cleared.
   - Players are offered the option to top up their balance.
   - Both players place their bets via console input.
4. **Card Dealing**:
   - Each player and the dealer receive two cards (dealer's second card is hidden).
5. **Player Actions (makeMove())**:
   - Each player views their hand and the dealer's visible card.
   - The user can enter "hit", "stand", or "double":
     - hit: draw one card
     - stand: stop drawing
     - double: double bet, draw once, and stop
6. **Dealer Turn**:
   - The dealer draws cards until reaching 17 (hard or soft, depending on the rules).
7. **Result Evaluation (displayResults())**:
   - Each player's hand is compared to the dealer's.
   - Output: win, loss, draw.
   - Balance is adjusted accordingly.
8. **Repeat Prompt**:
   - The system asks the user if they want to play again (y/n).
   - If y → new round starts; if n → program exits.

3.1 Output: Example of a Possible Game Round



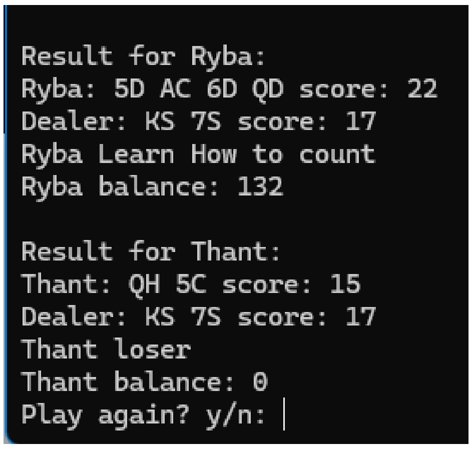
```
Welcome to Blackjack by MR I.Rybalko and Mr K.M.Thant

 Tutututuututututuut Begining
Ryba balance 100 Add money? y/n :y
Enter amount to add : 132
Ryba added 132 to balance/// Total: 232
Thant balance 100 Add money? y/n :n
```

1. Program welcomes players and begins a new round.
2. Players are asked if they want to add money.



```
Ryba enter your bet: 100
Thant enter your bet: 100
Ryba: 5D AC score: 16
Dealer's open card: KS
hit / stand: / double:hit
You got: 6D
Ryba: 5D AC 6D score: 12
Dealer's open card: KS
hit / stand:hit
You got: QD
Ryba: 5D AC 6D QD score: 22
Dealer's open card: KS
its not ypur money anymore
Thant: QH 5C score: 15
Dealer's open card: KS
hit / stand:stand
Dealer's turn
Dealer stands on 17
```

3. Both players place their bets.
4. Cards are dealt to all participants.
5. Each player takes action based on their hand and the dealer’s visible card.
6. The dealer plays automatically.



```

Result for Ryba:
Ryba: 5D AC 6D QD score: 22
Dealer: KS 7S score: 17
Ryba Learn How to count
Ryba balance: 132

Result for Thant:
Thant: QH 5C score: 15
Dealer: KS 7S score: 17
Thant loser
Thant balance: 0
Play again? y/n:
```

7. Final results and balances are displayed.

4. Suggestions for Improvement

1. **Graphical User Interface (GUI):**
   Replace the console interface with a GUI (e.g., using SFML, Qt, or a web interface). This will improve usability and presentation.
2. **Game Statistics Tracking:**
   Add tracking for number of games played, wins/losses per player, highest bet, etc.
3. **Card Animation or Visuals:**
   Even in console mode, adding simple ASCII card visuals would enhance the user experience.
4. **Persistent Balance Saving:**
   Store player balances between sessions using file I/O or a database.

5. Conclusion:

All the core features we wanted to see in the game were successfully implemented in code.
We initially chose this project not only because it was useful for practicing object-oriented programming, but also because it seemed fun and engaging.

While the final version still lacks some improvements required for a fully polished product, we believe that — as a first working game project in our portfolio — it turned out quite well.

Our team had fun writing this code as can be seen from some humorous lines , but despite the informal touches, the project was successfully completed.

6. Working program code:

Cards.h

```
#pragma once

#include <string>

using namespace std;

class Cards {

private:

    string RankCard;

    char Symbol;

public:

    Cards(string rankCard, char symbol);

    int GetValueFormCard() const;

    string CardSymbolPlusRank() const;

    string GetRankCard() const;

    char GetSymbol() const;

};
```

Card.cpp

```
#include "Cards.h"

Cards::Cards(string rankCard, char symbol) : RankCard(rankCard), Symbol(symbol) {}

int Cards::GetValueFormCard() const {
    if (RankCard == "A") return 11;
    if (RankCard == "K" || RankCard == "Q" || RankCard == "J") return 10;
    return stoi(RankCard);
}
string Cards::CardSymbolPlusRank() const {
    return RankCard + Symbol;
}
string Cards::GetRankCard() const {
    return RankCard;
}
char Cards::GetSymbol() const {
    return Symbol;
}
```

Deck.h

```
#pragma once
#include <vector>
#include "Cards.h"
using namespace std;

class Deck {
private:
    vector<Cards> cards;
public:
    Deck();
    void initialize();
    void shuffle();
    int size() const;
    Cards drawCard();
};
```

Deck.cpp

```
#include "Deck.h"
#include <random>
#include <algorithm>
#include <iostream>

Deck::Deck() {
    initialize();
    shuffle();
}
void Deck::initialize() {
    vector<string> ranks = { "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A" };
    vector<char> suits = { 'S', 'H', 'D', 'C' };
    cards.clear();
    for (const auto& rank : ranks)
        for (char suit : suits)
            cards.emplace_back(rank, suit);
}
void Deck::shuffle() {
    random_device rd;
    mt19937 g(rd());
    std::shuffle(cards.begin(), cards.end(), g);
}
int Deck::size() const {
    return static_cast<int>(cards.size());
}
Cards Deck::drawCard() {
    if (cards.empty()) {
        cerr << "error, deck is empty!" << endl;
        exit(1);
    }
    Cards top = cards.back();
    cards.pop_back();
    return top;
}
```

Hand.h

```
#pragma once
#include <vector>
#include "Cards.h"
using namespace std;

class Hand {
private:
    vector<Cards> cards;
public:
    void addCard(const Cards& card);
    void clear();
    int getScore() const;
    bool isBust() const;
    bool isSoft() const;
    void print() const;
    int size() const;
    Cards getFirstCard() const;
};
```

Hand.cpp

```
#include "Hand.h"
#include <iostream>
#include <stdexcept>

void Hand::addCard(const Cards& card) {
    cards.push_back(card);
}
```

```
void Hand::clear() {
    cards.clear();
}
bool Hand::isSoft() const {
    if (getScore() > 21) return false;
    for (const auto& c : cards)
        if (c.GetRankCard() == "A") return true;
    return false;
}
int Hand::getScore() const {
    int score = 0, aceCount = 0;
    for (const auto& card : cards) {
        score += card.GetValueFormCard();
        if (card.GetRankCard() == "A") aceCount++;
    }
    while (score > 21 && aceCount > 0) {
        score -= 10;
        aceCount--;
    }
    return score;
}
bool Hand::isBust() const {
    return getScore() > 21;
}
void Hand::print() const {
    for (const auto& card : cards)
        cout << card.CardSymbolPlusRank() << " ";
    cout << "(score: " << getScore() << ")" << endl;
}
int Hand::size() const {
    return static_cast<int>(cards.size());
}
```

```
Cards Hand::getFirstCard() const {
    if (!cards.empty()) return cards[0];
    throw runtime_error("Hand is empty");
}
```

Player.h

```
#pragma once
#include <string>
#include "Hand.h"
#include "Deck.h"
using namespace std;

class Player {
protected:
    string name;
    int balance;
    int currentBet;
    Hand hand;

public:
    Player(const string& name, int balance);
    virtual ~Player() = default;

    virtual void makeMove(const Cards& dealerCard, Deck& deck) = 0;
    void receiveCard(const Cards& card);
    void clearHand();
    int getScore() const;
    bool isBust() const;
    void placeBet(int amount);
    void adjustBalance(int amount);
    int getBet() const;
    int getBalance() const;
    string getName() const;
    void printHand() const;
    Cards getVisibleCard() const;
    void addMoney(int amount);
};
```

Player.cpp

```
#include "Player.h"
#include <iostream>

Player::Player(const string& name, int balance)
    : name(name), balance(balance), currentBet(0) {
}
void Player::receiveCard(const Cards& card) {
    hand.addCard(card);
}
void Player::clearHand() {
    hand.clear();
    currentBet = 0;
}
int Player::getScore() const {
    return hand.getScore();
}
bool Player::isBust() const {
    return hand.isBust();
}
void Player::placeBet(int amount) {
    if (amount > balance) amount = balance;
    currentBet += amount;
    balance -= amount;
}
void Player::adjustBalance(int amount) {
    balance += amount;
}
int Player::getBet() const {
    return currentBet;
}
int Player::getBalance() const {
    return balance;
}
string Player::getName() const {
    return name;
}
void Player::printHand() const {
    cout << name << ": ";
    hand.print();
}
Cards Player::getVisibleCard() const {
    return hand.getFirstCard();
}
void Player::addMoney(int amount) {
    if (amount > 0) {
        balance += amount;
        cout << name << " added " << amount << " to balance. Total: " << balance << endl;
    }
    else {
        cout << "Invalid amount.\n";
    }
}
```

PlayerHuman.h

```
#pragma once
#include "Player.h"

class PlayerHuman : public Player {
public:
    PlayerHuman(const string& name, int balance);
    void makeMove(const Cards& dealerCard, Deck& deck) override;
};
```

PlayerHuman.cpp

```
#include "PlayerHuman.h"
#include <iostream>

PlayerHuman::PlayerHuman(const string& name, int balance) : Player(name, balance) {}

void PlayerHuman::makeMove(const Cards& dealerCard, Deck& deck) {
    string action;
    bool firstMove = true;
    while (true) {
        printHand();
        cout << "Dealer's open card: " << dealerCard.CardSymbolPlusRank() << endl;

        if (isBust()) {
            cout << "You busted!" << endl;
            break;
        }
        cout << "(hit / stand";
        if (firstMove && getBalance() >= getBet()) cout << " / double";
        cout << "): ";
        cin >> action;
        if (action == "hit") {
            Cards newCard = deck.drawCard();
            cout << "You got: " << newCard.CardSymbolPlusRank() << endl;
            receiveCard(newCard);
        }
        else if (action == "stand") {
            break;
        }
        else if (action == "double" && firstMove && getBalance() >= getBet()) {
            placeBet(getBet());
            cout << "You doubled your bet to " << getBet() << endl;
            Cards newCard = deck.drawCard();
            cout << "You got: " << newCard.CardSymbolPlusRank() << endl;
            receiveCard(newCard);
            break;
        }
        else {
            cout << "Invalid input. Try again.\n";
            continue;
        }
        firstMove = false;
    }
}
```

Dealer.h

```
#pragma once
#include "Player.h"

constexpr int DEALER_STAND_HARD = 17;
constexpr int DEALER_STAND_SOFT = 17;

class Dealer : public Player {
    bool standOnSoft17;

public:
    Dealer(bool s17 = true);
    void makeMove(const Cards&, Deck& deck) override;
    static bool isSoft(const Hand& h);

private:
    void draw(Deck& d);
};
```

Dealer.cpp

```
#include "Dealer.h"
#include <iostream>

Dealer::Dealer(bool s17) : Player("Dealer", 0), standOnSoft17(s17) {}

bool Dealer::isSoft(const Hand& h) {
  return h.isSoft();
}

void Dealer::makeMove(const Cards&, Deck& deck) {
  std::cout << "Dealer's turn\n";
  while (true) {
    int score = getScore();
    bool soft = isSoft(hand);

    if (score < DEALER_STAND_HARD) {
      draw(deck);
    }
    else if (score == DEALER_STAND_SOFT && soft && !standOnSoft17) {
      draw(deck);
    }
    else {
      break;
    }
  }
  std::cout << "Dealer stands on " << getScore() << '\n';
}

void Dealer::draw(Deck& d) {
  Cards c = d.drawCard();
  receiveCard(c);
  std::cout << "Dealer draws " << c.CardSymbolPlusRank() << '\n';
}
```

System.h

```
#pragma once
#include "Deck.h"
#include "PlayerHuman.h"
#include "Dealer.h"

class System {
private:
    Deck deck;
    PlayerHuman player1;
    PlayerHuman player2;
    Dealer dealer;

public:
    System();
    void start();
    void playRound();
    void displayResults(PlayerHuman& player);
    void handleTopUp(PlayerHuman& player);
};
```

System.cpp

```
#include "System.h"
#include <iostream>

System::System() : player1("Ryba", 100), player2("Kyi", 100), dealer() {}

void System::start() {
    cout << "Welcome to Blackjack!\n";
    while (true) {
        if (player1.getBalance() <= 0 && player2.getBalance() <= 0) {
            cout << "Both players are out of money. Game over.\n";
            break;
        }
        playRound();
        char again;
        cout << "Play again? (y/n): ";
        cin >> again;
        if (again != 'y') break;
    }
}
void System::handleTopUp(PlayerHuman& player) {
    char choice;
    cout << player.getName() << " balance: " << player.getBalance() << ". Add money? (y/n): ";
    cin >> choice;
    if (choice == 'y') {
        int amount;
        cout << "Enter amount to add: ";
        cin >> amount;
        player.addMoney(amount);
    }
}
void System::playRound() {
    deck.initialize();
    deck.shuffle();
    player1.clearHand();
    player2.clearHand();
    dealer.clearHand();
    cout << "\n--- New Round ---\n";
    handleTopUp(player1);
    handleTopUp(player2);
    int bet1, bet2;
    cout << player1.getName() << ", enter your bet: ";
    cin >> bet1;
    player1.placeBet(bet1);
    cout << player2.getName() << ", enter your bet: ";
    cin >> bet2;
    player2.placeBet(bet2);
    player1.receiveCard(deck.drawCard());
    player2.receiveCard(deck.drawCard());
    dealer.receiveCard(deck.drawCard());
    player1.receiveCard(deck.drawCard());
    player2.receiveCard(deck.drawCard());
    dealer.receiveCard(deck.drawCard());

    player1.makeMove(dealer.getVisibleCard(), deck);
    player2.makeMove(dealer.getVisibleCard(), deck);
    dealer.makeMove(dealer.getVisibleCard(), deck);

    displayResults(player1);
    displayResults(player2);
}
void System::displayResults(PlayerHuman& player) {
    cout << "\nResult for " << player.getName() << ":\n";
    player.printHand();
    dealer.printHand();

    int playerScore = player.getScore();
    int dealerScore = dealer.getScore();

    if (player.isBust()) {
        cout << player.getName() << " busted.\n";
    }
    else if (dealer.isBust() || playerScore > dealerScore) {
        cout << player.getName() << " wins!\n";
        player.adjustBalance(player.getBet() * 2);
    }
    else if (playerScore == dealerScore) {
        cout << player.getName() << " draws.\n";
        player.adjustBalance(player.getBet());
    }
    else {
        cout << player.getName() << " loses.\n";
    }
    cout << player.getName() << " balance: " << player.getBalance() << "\n";
}
```

Main.cpp (Blackjack.cpp)

```
#include "System.h"

int main() {
    System game;
    game.start();
    return 0;
}
```