AI and Engine Pack: Card Games

Table of Contents

Introduction

This is the documentation and script reference for the AI and Engine pack: Card Games. Each game has one script file for the engine, and one for the AI. All of them have the necessary methods public.
In addition, the pack includes a demo script, that contains implementations of all of the 6 games. Attach the DemoScript to any game object or simply run the DemoScene to see them!

You are free to use demo games as is, or modify them as needed. However, the demo is meant for demonstration purposes, and thus does not contain all aspects of a full game such as animations.

Here are the rules for all of the games, in alphabetical order. The rules can also be found from the script reference and in the respective engine scripts.

Rules for Blackjack:

Each player is dealt two cards at the start and plays only against the house and not each other.
Players have 5 different actions to take: Stand, Hit, Double Down, Split or Surrender.
A player's turn continues until the player is standing, surrendered or their hand cards' total value is over 21.

The value of the hand is calculated so that ace is either 1 or 11 depending which is more beneficial to the player, cards over 10 have a value of 10 and every other card has their face value.

Stand means keeping the current cards.
Hit means getting a new card.
Double down means only getting one more card, but doubling the wager, and is only allowed on the first turn.
Split means splitting the cards into two new hands, both getting their own new card and wager, and is only allowed on the first turn if both cards have the same value.
Surrender means forfeiting the round and getting half of the wager back.

After all players are done, house keeps dealing itself cards until the total is over 16.
For all players standing with a total value of less than 22, if the house's value is lower than the player's or over 21, the player is paid according to their wager.
Having the same value as the house means getting the wager back as is.

Rules for Casino:

Each player is dealt four cards, as well as the table.
Players take turns to either use a card to collect cards from the table, or throw the card to the table.

To collect cards, the sum of the cards you collect must match the card you are using from your hand.
I.e. you can collect a six and a three from the table using a nine.
Players are also allowed to take multiples of the used hand card.
I.e. you can collect two sixes and two threes from the table using a nine.
It is not allowed to play a card but leave collectable cards on the table.

Some cards have special values when used from the hand (but not when they are on the table):
Two of spades has the value of 15.
Ten of diamonds has the value of 16.
Each ace has the value of 14.

When players are out of cards, each player are dealt 4 new cards.
This is repeated until the deck is empty, at which point the player who collected cards last gets the rest from the table.

Points are calculated at the end of each round:
2 points for collecting most spades,
1 point for collecting most cards,
2 points for collecting ten of diamonds,
1 point for collecting two of spades or any ace.
The first player to get 16 points wins.

Rules for GOPS (Game Of Pure Strategy):
Each player has cards from 1 to 13 that work as bids for a silent auction. Suits do not matter.
The players place hidden "bids" of a prize card, and the highest bid wins the card.

There are a total of 13 prize cards (from 1 to 13) and the current card is randomly selected from the remaining cards.

In case of a tie, the value of the prize is divided evenly among the highest bidders.
After all 13 cards have been auctioned on, the player with the highest amount of points wins.

Rules for Hearts:

The deck is dealt evenly among the players.
At the start, each player passes 3 cards to the player on the left, next round to the next player and so on.
After card passing, the player with the two of clubs starts.

Players take clockwise turns in placing one card on the table each trick.
The card has to be of the same suit as the one who started the trick.
A trick ends when all players have placed a card, and the player who placed the highest card of the original suit collects the cards and starts the next trick.

Hearts cannot be used to start a trick until at least one heart is played.
At the end of the round, players get score based on the cards they collected.

Each heart collected is +1 point, and the queen of spades is +13 points.
If one player collected all of the aforementioned cards, they get -26 points or other players get +26 points instead, depending which benefits them most.

When a player reaches a score of 100, the player with the lowest score wins.

Rules for Ninety Nine:
Each player is dealt 3 cards, and take turns in playing a card to the table.
After playing a card, the player gets a new card.

Only the total value of the table cards is relevant, and suits do not matter.
If a player makes the table’s total value go to over 99, that player loses a token.

Each player has three tokens at the start of the game, and drops out when the last token is lost.
Winner is the last person in the game.

Some cards have values that are different than the face value of the card:

Ace is either 1 or 11, decided by the player.
3 has the value 3 but the next player loses their turn.
4 has a value of 0 and the order of play is reversed. If there are only two players, the player gets another turn instead.
9 changes the total value of the table cards to 99 regardless of what it was previously.
10 is either -10 or +10, decided by the player.
11 and 12 have a value of 10.
13 has a value of 0.

Rules for Top Trumps:
The game can use any kind of cards. Not just classic playing cards.
Each card in the deck has same attributes, but with different values.
The deck is divided among all players as evenly as possible, and all players are only able to see the topmost card of their own cards.
First player initiates the first round, and from there, each round is initiated by the winner of the previous round.
The player initiating a round picks one of the attributes that the cards have, and then everyone plays their card on the table.
The player with the card that has the highest value in the picked attribute wins the round and gets the cards from other players.
In case of a tie, the cards go to the side instead and the next round is a tie-breaker, where the winner of that round gets all the cards from the previous round.
If a player does not have cards to take part in a tie-breaker, they lose automatically.
When a player runs out of cards from their hand, they shuffle the pile that they have accumulated through wins.
When a player is truly out of cards, they are out of the game.
The player who collects the entire deck, wins.

Rules for Sevens:
The deck is divided among all players. Some players may get one card more or less than others.
The player with the seven of hearts starts by placing the card on the table.

Players take clockwise turns in either placing a card on the table or passing their turn.
Passing is only allowed if the player cannot play any card.
Each suit is played separately.

Card with a face value of 7 is always playable.

Cards with a face value of less than 7 is only playable if the card with one higher face value has been played.
Cards with a face value of more than 7 is only playable if the card with one smaller face value has been played.
The winner is the player who plays their last card first.

Script Reference

Card Games AI and Engine

Deck.cs

Namespaces

CGAIE

```
namespace CGAIE
```

Enumerations

Suit

```
public enum Suit{
     club = 1,
     diamond = 2,
     heart = 3,
     spade = 4}
```

Enumeration for card suits.

Classes

Card

```
public class Card
```

Class implementing a playing card.

Constructors

Card

```
public  Card(
     Suit _suit,
     ushort _value)
```

Constructor for a card.

_suit: Suit as an enumeration. On of the four traditional playing card suits.
_value: Value of the card. From 1 to 13.

Operators

operator ==

```
public static bool operator ==(
     Card a,
     Card b)
```

operator !=

```
public static bool operator !=(
     Card a,
     Card b)
```

Variables

suit

```
public readonly Suit suit
```

Suit as an enumeration. On of the four traditional playing card suits.

value

```
public readonly ushort value
```

Value of the card. From 1 to 13.

Methods

Equals

```
public override bool Equals(
     object obj)
```

GetHashCode

```
public override int GetHashCode()
```

ToString

```
public override string ToString()
```

Displays the card as a string in the format "suit value"

sortValue

```
public int sortValue(
     bool ace14 = false)
```

Returns the sort value of the card. The sort value is a positive integer, going from 1 to 13 or 2 to 14 for each suit. The suits are in order: clubs, diamonds, hearts, spades.

ace14: Boolean to specify if ace is considered 14 instead of 1. False by default.

Returns: The sort value of the card as an integer.

Deck

```
public class Deck
```

Class implementing a typical deck of 52 playing cards.

Constructors

Deck

```
public  Deck(
     bool empty = false)
```

Create a new deck. Populated with normal 52 playing cards by default.

empty: Optionally the deck can be empty.

Properties

Cards

```
public List<Card> Cards
     { get;
     }
```

List of the cards in the deck

Methods

Deal

```
public Card Deal()
```

Deals one random card from the deck.

Returns: A random card from the deck. Null if the deck is empty.

CustomCard

```
public class CustomCard
```

Class implementing a custom card.

Constructors

CustomCard

```
public  CustomCard(
     uint _id,
     Dictionary<string, float> _attributes)
```

Constructor for a custom card.

_id: Unique identifier of the card. This id can be used to match the correct sprite. Lowest ID should be 1.
_attributes: Dictionary of the card’s attributes. The keys should match the list of attributes in the relevant deck.

Operators

operator ==

```
public static bool operator ==(
     CustomCard a,
     CustomCard b)
```

operator !=

```
public static bool operator !=(
     CustomCard a,
     CustomCard b)
```

Variables

ID

```
public readonly uint ID
```

Unique identifier of the card. This id can be used to match the correct sprite. Lowest ID should be 1.

attributes

```
public readonly Dictionary<string,float> attributes
```

Dictionary of the card’s attributes. The keys should match the list of attributes in the relevant deck.

Methods

Equals

```
public override bool Equals(
     object obj)
```

GetHashCode

```
public override int GetHashCode()
```

ToString

```
public override string ToString()
```

CustomDeck

```
public class CustomDeck
```

Class implementing any kind of deck with any amount of cards and thier attributes.

Constructors

CustomDeck

```
public  CustomDeck(
     string[] cardAttributes)
```

Constructor for a custom deck. The deck is empty at the start. Use AddCard()-method to add cards.

Variables

CardAttributes

```
public readonly string[] CardAttributes
```

List of all the attributes that the Custom Cards in this deck have.

Properties

Cards

```
public List<CustomCard> Cards
      { get;
      }
```

List of the CustomCards in this deck.

Methods

AddCard

```
public bool AddCard(
      CustomCard card)
```

Adds the given card in the deck if it contains all required attributes.

card: Custom Card to be added to the deck

Returns: True if the card had all the required attributes. False otherwise.

Deal

```
public CustomCard Deal()
```

Deals one random card from the deck.

Returns: A random card from the deck. Null if the deck is empty.

Blackjack

C# BlackjackAI.cs

Namespaces

CGAIE

```
namespace CGAIE
```

Classes

BlackjackAI

```
public class BlackjackAI
```

AI for the card game Blackjack.

Enumerations

BlackjackAIType

```
public enum BlackjackAIType{
     random = 0,
     easy = 1,
     medium = 2,
     hard = 3}
```

Enumeration to define the difficulty of the AI.

Variables

AIType

```
public static BlackjackAIType AIType
```

The type of the AI. As a BlackjackAIType enumeration.

Methods

MakeMove

```
public static BlackjackEngine.BlackjackMove MakeMove(
     List<Card> hand,
     BlackjackEngine.HandState handState,
     Card dealerUpCard)
```

The AI makes a move using the specified AI type.

hand: List of current player's hand cards.
handState: The state of the current player's hand.
dealerUpCard: The dealer's upcard.

Returns: Blackjack move as a BlackjackEngine.BlackjackMove.

C# BlackjackEngine.cs

Namespaces

CGAIE

```
namespace CGAIE
```

Classes

BlackjackEngine

```
public static class BlackjackEngine
```

Engine for the card game Blackjack. Rules:

Each player is dealt two cards at the start and plays only against the house and not each other. Players have 5 different actions to take: Stand, Hit, Double Down, Split or Surrender. A player’s turn continues until the player is standing, surrendered or their hand cards’ total value is over 21.

The value of the hand is calculated so that ace is either 1 or 11 depending which is more beneficial to the player, cards over 10 have a value of 10 and every other card has their face value.

Stand means keeping the current cards. Hit means getting a new card. Double down means only getting one more card, but doubling the wager, and is only allowed on the first turn. Split means splitting the cards into two new hands, both getting their own new card and wager, and is only allowed on the first turn if both cards have the same value. Surrender means forfeiting the round and getting half of the wager back.

After all players are done, house keeps dealing itself cards until the total is over 16. For all players standing with a total value of less than 22, if the house’s value is lower than the player’s or over 21, the player is paid according to their wager. Having the same value as the house means getting the wager back as is.

Enumerations

GameState

```
public enum GameState{
     inactive = 0,
     normalTurn = 1,
     cardDealing = 2,
     endOfTurn = 3}
```

Enumeration to specify the current state of the game.

inactive: The game is not active. Execute ResetGame()-method to start it.

normalTurn: It is a player’s turn.

cardDealing: The dealer is dealing cards

endOfTurn: The game is between turns.

PlayerType

```
public enum PlayerType{
     local = 0,
     AI = 1,
     custom = 2}
```

Enumeration to specify the type of a player.

local: Local player. The input is given through the LocalMove()-method.

AI: AI. The move is made automatically by the AI.

custom: Custom. The input is given through the CustomMove()-method. Can be used, for example, to input the moves of a remote player.

BlackjackMove

```
public enum BlackjackMove{
     stand = 0,
     hit = 1,
     doubleDown = 2,
     split = 3,
     surrender = 4}
```

Enumeration to represent a move in Blackjack.

stand: Keep the current total.

hit: Get a new card.

doubleDown: Get only one new card, but double the wager.

split: Split the cards into two new hands.

surrender: Forfeit the round and get half of your wager back.

HandState

```
public enum HandState{
     fresh = 0,
     doubled = 1,
     split = 2,
     surrendered = 3,
     standing = 4,
     playing = 5,
     bust = 6}
```

fresh: The hand is freshly dealt.

doubled: The hand was just doubled.

split: The hand was just split.

surrendered: The hand has surrendered.

standing: The hand is standing.

playing: The hand is playing

bust: The hand is bust (value over 21).

Variables

waitPeriod

```
public static int waitPeriod
```

time to wait between certain actions in milliseconds

wager

```
public static int wager
```

The base wager.

startingChips

```
public static int startingChips
```

The amount of chips every player starts with.

Properties

seats

```
public static PlayerType[] seats
     { get;
     set; }
```

Number of players. From 1 to 6

Hands

```
public static List<Card>[] Hands
     { get;
     }
```

The players’ hand cards.

SecondHands

```
public static List<Card>[] SecondHands
     { get;
     }
```

The players’ cards on the "secondary" split hand.

Table

```
public static List<Card> Table
     { get;
     }
```

The house’s cards

Chips

```
public static int[] Chips
     { get;
     }
```

Each player’s chips.

CurrentPlayer

```
public static int CurrentPlayer
     { get;
     }
```

The current player, ranges from 1 to length of ’seats’.

Bets

```
public static int[] Bets
     { get;
     }
```

The wager for this round for each player (on their main hand).

SecondaryBets

```
public static int[] SecondaryBets
     { get;
     }
```

The wager for this round for each player on the secondary, split, hand.

HandStates

```
public static HandState[] HandStates
     { get;
     }
```

State of the player’s (main) hand.

SecondHandStates

```
public static HandState[] SecondHandStates
     { get;
     }
```

State of the player’s secondary, split, hand.

gameState

```
public static GameState gameState
     { get;
     }
```

The current state of the game. As a GameState enumeration.

Methods

ResetGame

```
public static void ResetGame(
     bool resetChips = true)
```

Resets all variables related to the current game and stops the AI if needed.

StopAI

```
public static void StopAI()
```

This method can be used to stop the AI. This will abort the whole AI thread, meaning that the AI will not do a move. Should not be needed by the user.

LocalMove

```
public static void LocalMove(
     BlackjackMove move)
```

Executes the given move if the current player is a local player.

move: The Blackjackmove you want to execute.

CustomMove

```
public static void CustomMove(
     BlackjackMove move)
```

Executes the given move if the current player is a custom player.

move: The Blackjackmove you want to execute.

PossibleMoves

```
public static List<BlackjackMove> PossibleMoves(
     bool mainHand = true)
```

Finds all possible moves for the current player.

Returns: List of all possible moves for the current player as BlackJackMove.

PossibleMoves

```
public static List<BlackjackMove> PossibleMoves(
     List<Card> handCards,
     HandState handState)
```

Finds all possible options for the given hand cards.

Returns: List of all possible moves for the given parameters as BlackjackMoves.

HandTotalValue

```
public static int HandTotalValue(
     List<Card> handCards)
```

Calculates the total value of the given hand.

handCards: List of cards in the hand.

Returns: The total value (in terms of Blackjack) of the hand.

Casino

C# CasinoAI.cs

Namespaces

CGAIE

```
namespace CGAIE
```

Classes

CasinoAI

```
public static class CasinoAI
```

AI for the card game Casino.

Enumerations

CasinoAIType

```
public enum CasinoAIType{
     random = 0,
     easy = 1,
     medium = 2,
     hard = 3}
```

Enumeration to define the difficulty of the AI.

Variables

AIType

```
public static CasinoAIType AIType
```

The type of the AI. As a HeartsAIType enumeration.

Methods

MakeMove

```
public static CasinoEngine.CasinoMove MakeMove(
     List<Card> hand,
     List<Card> table,
     List<Card>[] collected)
```

The AI makes a move using the specified AI type.

hand: List of current player’s hand cards.
table: List of the cards on the table.
collected: Array of lists of all players’ collected cards.

Returns: Casino move as a CasinoEngine.CasinoMove.

MakeMoveRandom

```
public static CasinoEngine.CasinoMove MakeMoveRandom(
     List<CasinoEngine.CasinoMove> possibleMoves)
```

MakeMoveEasy

```
public static CasinoEngine.CasinoMove MakeMoveEasy(
     List<CasinoEngine.CasinoMove> possibleMoves)
```

MakeMoveMedium

```
public static CasinoEngine.CasinoMove MakeMoveMedium(
     List<CasinoEngine.CasinoMove> possibleMoves)
```

MakeMoveHard

```
public static CasinoEngine.CasinoMove MakeMoveHard(
     List<CasinoEngine.CasinoMove> possibleMoves,
     List<Card>[] collected)
```

C# CasinoEngine.cs

Namespaces

CGAIE

```
namespace CGAIE
```

Classes

CasinoEngine

```
public static class CasinoEngine
```

Engine for the card game Casino. Rules:

Each player is dealt four cards, as well as the table. Players take turns to either use a card to collect cards from the table, or throw the card to the table.

To collect cards, the sum of the cards you collect must match the card you are using from your hand. I.e. you can collect a six and a three from the table using a nine. Players are also allowed to take multiples of the used hand card. I.e. you can collect two sixes and two threes from the table using a nine. It is not allowed to play a card but leave collectable cards on the table.

Some cards have special values when used from the hand (but not when they are on the table): Two of spades has the value of 15. Ten of diamonds has the value of 16. Each ace has the value of 14.

When players are out of cards, each player are dealt 4 new cards. This is repeated until the deck is empty, at which point the player who collected cards last gets the rest from the table.

Points are calculated at the end of each round: 2 points for collecting most spades, 1 point for collecting most cards, 2 points for collecting ten of diamonds, 1 point for collecting two of spades or any ace. The first player to get 16 points wins.

Enumerations

GameState

```
public enum GameState{
     inactive = 0,
     cardDealing = 1,
     normalTurn = 2,
     endOfTurn = 3}
```

Enumeration to specify the current state of the game.

inactive: The game is not active. Execute ResetGame()-method to start it.

cardDealing: Cards are being dealt.

normalTurn: It is a player’s turn.

endOfTurn: The game is between turns.

PlayerType

```
public enum PlayerType{
     local = 0,
     AI = 1,
     custom = 2}
```

Enumeration to specify the type of a player.

local: Local player. The input is given through the LocalMove()-method.

AI: AI. The move is made automatically by the AI.

custom: Custom. The input is given through the CustomMove()-method. Can be used, for example, to input the moves of a remote player.

Classes

CasinoMove

```
public struct CasinoMove
```

A struct to represent moves in Casino. A move consists of a hand card and possible list of cards collected from the table.

Constructors

CasinoMove

```
public  CasinoMove(
     Card _handCard,
     List<Card> _tableCards)
```

CasinoMove

```
public  CasinoMove(
     Card _handCard,
     Card[] _tableCards)
```

Variables

handCard

```
public readonly Card handCard
```

tableCards

```
public readonly List<Card> tableCards
```

Variables

waitPeriod

```
public static int waitPeriod
```

time to wait between certain actions in milliseconds

Properties

seats

```
public static PlayerType[] seats
     { get;
     set; }
```

Number of players. From 2 to 4

Score

```
public static int[] Score
     { get;
     }
```

Each player’s score.

Hands

```
public static List<Card>[] Hands
     { get;
     }
```

List of cards on each player’s hand.

Collected

```
public static List<Card>[] Collected
     { get;
     }
```

List of cards collected by each player.

Table

```
public static List<Card> Table
     { get;
     }
```

List of cards on the table.

CurrentPlayer

```
public static int CurrentPlayer
     { get;
     }
```

The current player. Ranges from 1 to length of ’seats’.

UsedCard

```
public static Card UsedCard
     { get;
     }
```

The hand card that was just used by a player.

TakenCards

```
public static List<Card> TakenCards
     { get;
     }
```

The cards just collected from the table by a player.

cardsInDeck

```
public static int cardsInDeck
     { get;
     }
```

Number of cards left in the deck.

gameState

```
public static GameState gameState
     { get;
     }
```

The current state of the game. As a GameState enumeration.

Methods

ResetGame

```
public static void ResetGame()
```

Resets all variables related to the current game and stops the AI if needed.

ResetRound

```
public static void ResetRound()
```

Resets all variables related to the current round and stops the AI if needed.

StopAI

```
public static void StopAI()
```

This method can be used to stop the AI. This will abort the whole AI thread, meaning that the AI will not do a move. Should not be needed by the user.

LocalMove

```
public static void LocalMove(
     Card card,
     List<Card> tableCards)
```

Executes the given move if the current player is a local player.

card: The used hand card.
tableCards: List of possibly collected table cards.

CustomMove

```
public static void CustomMove(
     Card card,
     List<Card> tableCards)
```

Executes the given move if the current player is a custom player.

card: The used hand card.
tableCards: List of possibly collected table cards.

CardCombinations

```
public static IEnumerable<Card[]> CardCombinations(
     Card[] cards)
```

Find all combinations of the given cards. Useful for finding out what a player can collect from the table.

cards: An array of cards.

Returns: An enumerable for all the combinations of cards in the given array.

PossibleMoves

```
public static List<CasinoMove> PossibleMoves(
     Card card,
     List<Card> tableCards)
```

Finds all possible moves for given card and list of cards on the table.

card: Card to use.
tableCards: List of cards on the table.

Returns: List of possible casino moves for the given cards.

PossibleMoves

```
public static List<CasinoMove> PossibleMoves()
```

Finds all possible moves for the current player

Returns: List of all possible moves for the current player as CasinoMoves

PossibleMoves

```
public static List<CasinoMove> PossibleMoves(
     List<Card> handCards,
     List<Card> tableCards)
```

Finds all possible moves for the given hand cards and table

Returns: List of all possible moves for the given parameters as CasinoMoves

GOPS

C# GOPSAI.cs

Namespaces

CGAIE

```
namespace CGAIE
```

Classes

GOPSAI

```
public static class GOPSAI
```

AI for the card game GOPS (Game Of Pure Strategy).

Enumerations

GOPSAIType

```
public enum GOPSAIType{
     random = 0,
     easy = 1,
     medium = 2,
     hard = 3}
```

Enumeration to define the difficulty of the AI.

Variables

AIType

```
public static GOPSAIType AIType
```

The type of the AI. As a GOPSAIType enumeration.

Methods

MakeMove

```
public static Card MakeMove(
     List<Card> hand,
     Card table,
     Card previousPrize,
     Card[] previousBids,
     int currentPlayerIndex)
```

The AI makes a move using the specified AI type.

hand: List of current player’s hand cards.
table: The current card prize on the table.
previousPrize: The previous prize card.
previousBids: List of the previous prize’s bids.
currentPlayerIndex: The index of this player. I.e. which bid in previousBids is this player’s.

Returns: GOPS move as a Card.

C# GOPSEngine.cs

Namespaces

CGAIE

```
namespace CGAIE
```

Classes

GOPSEngine

```
public static class GOPSEngine
```

Engine for the card game GOPS (Game Of Pure Strategy). Rules:

Each player has cards from 1 to 13 that work as bids for a silent auction. Suits do not matter. The players place hidden "bids" of a prize card, and the highest bid wins the card.

There are a total of 13 prize cards (from 1 to 13) and the current card is randomly selected from the remaining cards.

In case of a tie, the value of the prize is divided evenly among the highest bidders. After all 13 cards have been auctioned on, the player with the highest amount of points wins.

Enumerations

GameState

```
public enum GameState{
     inactive = 0,
     normalTurn = 1,
     endOfTurn = 2}
```

Enumeration to specify the current state of the game.

inactive: The game is not active. Execute ResetGame()-method to start it.

normalTurn: It is a player’s turn.

endOfTurn: The game is between turns.

PlayerType

```
public enum PlayerType{
     local = 0,
     AI = 1,
     custom = 2}
```

Enumeration to specify the type of a player.

local: Local player. The input is given through the LocalMove()-method.

AI: AI. The move is made automatically by the AI.

custom: Custom. The input is given through the CustomMove()-method. Can be used, for example, to input the moves of a remote player.

Variables

waitPeriod

```
public static int waitPeriod
```

time to wait between rounds in milliseconds

Properties

seats

```
public static PlayerType[] seats
     { get;
     set; }
```

Number of players. From 2 to 6

Hands

```
public static List<Card>[] Hands
     { get;
     }
```

List of hand cards for each player.

Table

```
public static List<Card> Table
     { get;
     }
```

List of cards on the table. The first card in the list is the current prize.

Points

```
public static float[] Points
     { get;
     }
```

Array of points for each player.

CurrentPlayer

```
public static int CurrentPlayer
     { get;
     }
```

The current player’s number. Ranges from 1 to length of ’seats’.

Bids

```
public static Card[] Bids
     { get;
     }
```

Array of bids from all players for the current prize.

gameState

```
public static GameState gameState
     { get;
     }
```

The current state of the game. As a GameState enumeration.

Methods

ResetGame

```
public static void ResetGame()
```

Resets all variables related to the current game and stops the AI if needed.

StopAI

```
public static void StopAI()
```

This method can be used to stop the AI. This will abort the whole AI thread, meaning that the AI will not do a move. Should not be needed by the user.

LocalMove

```
public static void LocalMove(
     Card card)
```

Executes the given move if the current player is a local player.

card: The used hand card.

CustomMove

```
public static void CustomMove(
     Card card)
```

Executes the given move if the current player is a custom player.

card: The used hand card.

PossibleMoves

```
public static List<Card> PossibleMoves()
```

Finds all possible moves for the current player

Returns: List of all possible moves for the current player as Cards

PossibleMoves

```
public static List<Card> PossibleMoves(
     List<Card> handCards)
```

Finds all possible moves for the given hand cards

Returns: List of all possible moves for the given parameters as GOPSMoves

Hearts

C# HeartsAI.cs

Namespaces

CGAIE

```
namespace CGAIE
```

Classes

HeartsAI

```
public static class HeartsAI
```

AI for the card game Hearts.

Enumerations

HeartsAIType

```
public enum HeartsAIType{
     random = 0,
     easy = 1,
     medium = 2,
     hard = 3}
```

Enumeration to define the difficulty of the AI.

Variables

AIType

```
public static HeartsAIType AIType
```

The type of the AI. As a HeartsAIType enumeration.

Methods

MakeMove

```
public static Card MakeMove(
     List<Card> hand,
     List<Card>[] collected,
     List<Card> table,
     HeartsEngine.GameState gs)
```

The AI makes a move using the specified AI type.

hand: List of current player’s hand cards.
collected: Array of lists of cards collected by each player.
table: The cards that have been played for the current trick already.
gs: The state of the game. I.e. should the AI pass cards or is it a normal turn.

Returns: Hearts move as a Card.

MakeMoveRandom

```
public static Card MakeMoveRandom(
     List<Card> possibleMoves)
```

MakeMoveEasy

```
public static Card MakeMoveEasy(
     List<Card> possibleMoves,
     HeartsEngine.GameState gs,
     List<Card> table)
```

MakeMoveMedium

```
public static Card MakeMoveMedium(
     List<Card> possibleMoves,
     HeartsEngine.GameState gs,
     List<Card> table,
     List<Card>[] collected)
```

MakeMoveHard

```
public static Card MakeMoveHard(
     List<Card> possibleMoves,
     HeartsEngine.GameState gs,
     List<Card> table,
     List<Card>[] collected)
```

C# HeartsEngine.cs

Namespaces

CGAIE

```
namespace CGAIE
```

Classes

HeartsEngine

```
public static class HeartsEngine
```

Engine for the card game Hearts. Rules:

The deck is dealt evenly among the players. At the start, each player passes 3 cards to the player on the left, next round to the next player and so on. After card passing, the player with the two of clubs starts.

Players take clockwise turns in placing one card on the table each trick. The card has to be of the same suit as the one who started the trick. A trick ends when all players have placed a card, and the player who placed the highest card of the original suit collects the cards and starts the next trick.

Hearts cannot be used to start a trick until at least one heart is played. At the end of the round, players get score based on the cards they collected.

Each heart collected is +1 point, and the queen of spades is +13 points. If one player collected all of the aforementioned cards, they get -26 points or other players get +26 points instead, depending which benefits them most.

When a player reaches a score of 100, the player with the lowest score wins.

Enumerations

GameState

```
public enum GameState{
     inactive = 0,
     cardPassing = 1,
     normalTurn = 2}
```

Enumeration to specify the current state of the game.

inactive: The game is not active. Execute ResetGame()-method to start it.

PlayerType

```
public enum PlayerType{
     local = 0,
     AI = 1,
     custom = 2}
```

Enumeration to specify the type of a player.

local: Local player. The input is given through the LocalMove()-method.

AI: AI. The move is made automatically by the HeartsAI class.

custom: Custom. The input is given through the CustomMove()-method. Can be used, for example, to input the moves of a remote player.

Variables

waitPeriod

```
public static int waitPeriod
```

time to wait between certain actions in milliseconds

Properties

seats

```
public static PlayerType[] seats
     { get;
     set; }
```

Number of players. From 3 to 6

Score

```
public static int[] Score
     { get;
     }
```

Array of each player’s score.

Hands

```
public static List<Card>[] Hands
     { get;
     }
```

Array of lists of cards in each player’s hand.

Collected

```
public static List<Card>[] Collected
     { get;
     }
```

Array of lists of cards each player has collected.

Table

```
public static List<Card> Table
     { get;
     }
```

List of cards already on the table for the current trick.

CurrentPlayer

```
public static int CurrentPlayer
     { get;
     }
```

The number of the current player. Ranges from 1 to length of ’seats’.

PassDifference

```
public static int PassDifference
     { get;
     }
```

The difference in player number for who to pass cards. Clockwise. If PassDifference is 0, no cards are passed.

gameState

```
public static GameState gameState
     { get;
     }
```

The current state of the game. As a GameState enumeration.

Methods

ResetGame

```
public static void ResetGame()
```

Resets all variables related to the current game and stops the AI if needed.

ResetRound

```
public static void ResetRound()
```

Resets all variables related to the current round and stops the AI if needed.

StopAI

```
public static void StopAI()
```

This method can be used to stop the AI. This will abort the whole AI thread, meaning that the AI will not do a move. Should not be needed by the user.

LocalMove

```
public static void LocalMove(
     Card card)
```

Executes the given move if the current player is a local player.

card: The used hand card.

CustomMove

```
public static void CustomMove(
     Card card)
```

Executes the given move if the current player is a custom player.

card: The used hand card.

PossibleMoves

```
public static List<Card> PossibleMoves(
     List<Card> hand,
     List<Card>[] collected,
     List<Card> table,
     GameState gs)
```

Finds all possible moves for the given situation.

hand: List of cards in the player's hand.
collected: Lists of cards collected by all players.
table: List of cards currently on the table.
gs: State of the game. I.e. card passing or normal turn.

Returns: List of cards that can be legitimately played from the player's hand.

PossibleMoves

```
public static List<Card> PossibleMoves()
```

Finds all possible moves for the current player.

Returns: List of cards that can be legitimately played from the current player's hand.

Ninety-nine

NinetyNineAI.cs

Namespaces

CGAIE

```
namespace CGAIE
```

Classes

NinetyNineAI

```
public static class NinetyNineAI
```

AI for the card game Ninety Nine.

Enumerations

NinetyNineAIType

```
public enum NinetyNineAIType{
     random = 0,
     easy = 1,
     medium = 2,
     hard = 3}
```

Enumeration to define the difficulty of the AI.

Variables

AIType

```
public static NinetyNineAIType AIType
```

The type of the AI. As a NinetyNineAIType enumeration.

Methods

MakeMove

```
public static NinetyNineEngine.NinetyNineMove MakeMove(
     List<Card> hand,
     int table)
```

The AI makes a move using the specified AI type.

hand: List of current player’s hand cards.
table: The current value of the pile of cards on the table.

Returns: Ninety Nine move as a NinetyNineEngine.NinetyNineMove.

C# NinetyNineEngine.cs

Namespaces

CGAIE

```
namespace CGAIE
```

Classes

NinetyNineEngine

```
public static class NinetyNineEngine
```

Engine for the card game Ninety Nine. Rules:

Each player is dealt 3 cards, and take turns in playing a card to the table. After playing a card, the player gets a new card.

Only the total value of the table cards is relevant, and suits do not matter. If a player makes the table's total value go to over 99, that player loses a token.

Each player has three tokens at the start of the game, and drops out when the last token is lost. Winner is the last person in the game.

Some cards have values that are different than the face value of the card: Ace is either 1 or 11, decided by the player. 3 has the value 3 but the next player loses their turn. 4 has a value of 0 and the order of play is reversed. If there are only two players, the player gets another turn instead. 9 changes the total value of the table cards to 99 regardless of what it was previously. 10 is either -10 or +10, decided by the player. 11 and 12 have a value of 10. 13 has a value of 0.

Enumerations

GameState

```
public enum GameState{
     inactive = 0,
     normalTurn = 1,
     cardDealing = 2,
     endOfTurn = 3}
```

Enumeration to specify the current state of the game.

inactive: The game is not active. Execute ResetGame()-method to start it.

normalTurn: It is a player's turn to play a card

cardDealing: Cards are being dealt to players

endOfTurn: It is the end of a player's turn

PlayerType

```
public enum PlayerType{
     local = 0,
     AI = 1,
     custom = 2}
```

Enumeration to specify the type of a player.

local: Local player. The input is given through the LocalMove()-method.

AI: AI. The move is made automatically by the AI.

custom: Custom. The input is given through the CustomMove()-method. Can be used, for example, to input the moves of a remote player.

Classes

NinetyNineMove

```
public class NinetyNineMove
```

A class to represent a move in Ninety Nine. A move consists of a card and its value.

Constructors

NinetyNineMove

```
public  NinetyNineMove(
     Card _card,
     int _value)
```

Variables

card

```
public readonly Card card
```

value

```
public readonly int value
```

Variables

waitPeriod

```
public static int waitPeriod
```

time to wait between certain actions in milliseconds

Properties

seats

```
public static PlayerType[] seats
     { get;
     set; }
```

Number of players. From 2 to 6

Hands

```
public static List<Card>[] Hands
     { get;
     }
```

Lists of cards in each player’s hand.

Table

```
public static int Table
     { get;
     }
```

The value of the pile of cards on the table.

Tokens

```
public static int[] Tokens
     { get;
     }
```

Number of tokens left for each player.

CurrentPlayer

```
public static int CurrentPlayer
     { get;
     }
```

The number of the current player. Ranges from 1 to length of ’seats’.

ClockwisePlay

```
public static bool ClockwisePlay
      { get;
      }
```

Boolean specifying if the game is currently played clockwise. Can be changed by playing a four of any suit.

UsedMove

```
public static NinetyNineMove UsedMove
      { get;
      }
```

The card used most recently

gameState

```
public static GameState gameState
      { get;
      }
```

The current state of the game. As a GameState enumeration.

Methods

ResetGame

```
public static void ResetGame()
```

Resets all variables related to the current game and stops the AI if needed.

StopAI

```
public static void StopAI()
```

This method can be used to stop the AI. This will abort the whole AI thread, meaning that the AI will not do a move. Should not be needed by the user.

LocalMove

```
public static void LocalMove(
      Card card,
      int value)
```

LocalMove

```
public static void LocalMove(
     NinetyNineMove move)
```

Executes the given move if the current player is a local player.

move: The move by the player.

CustomMove

```
public static void CustomMove(
     NinetyNineMove move)
```

Executes the given move if the current player is a custom player.

move: The move by the player.

PossibleMoves

```
public static List<NinetyNineMove> PossibleMoves()
```

Finds all possible moves for the current player

Returns: List of all possible moves for the current player as Cards

PossibleMoves

```
public static List<NinetyNineMove> PossibleMoves(
     List<Card> handCards)
```

Finds all possible moves for the given hand cards

Returns: List of all possible moves for the given parameters as NinetyNineMoves

Sevens

C# SevensAI.cs

Namespaces

CGAIE

```
namespace CGAIE
```

Classes

SevensAI

```
public static class SevensAI
```

AI for the card game Sevens.

Enumerations

SevensAIType

```
public enum SevensAIType{
     random = 0,
     medium = 1,
     hard = 2}
```

Enumeration to define the difficulty of the AI.

Variables

AIType

```
public static SevensAIType AIType
```

The type of the AI. As a SevensAIType enumeration.

Methods

MakeMove

```
public static Card MakeMove(
     List<Card> hand,
     List<Card> table)
```

The AI makes a move using the specified AI type.

hand: List of current player’s hand cards.
table: List of cards already on the table..

Returns: Sevens move as a Card.

MakeMoveRandom

```
public static Card MakeMoveRandom(
     List<Card> possibleMoves)
```

MakeMoveMedium

```
public static Card MakeMoveMedium(
     List<Card> possibleMoves)
```

MakeMoveHard

```
public static Card MakeMoveHard(
     List<Card> possibleMoves,
     List<Card> hand)
```

C# SevensEngine.cs

Namespaces

CGAIE

```
namespace CGAIE
```

Classes

SevensEngine

```
public static class SevensEngine
```

Engine for the card game Sevens. Rules:

The deck is divided among all players. Some players may get one card more or less than others. The player with the seven of hearts starts by placing the card on the table.

Players take clockwise turns in either placing a card on the table or passing their turn. Passing is only allowed if the player cannot play any card. Each suit is played separately.

Card with a face value of 7 is always playable. Cards with a face value of less than 7 is only playable if the card with one higher face value has been played. Cards with a face value of more than 7 is only playable if the card with one smaller face value has been played.

The winner is the player who plays their last card first.

Enumerations

GameState

```
public enum GameState{
     inactive = 0,
     normalTurn = 1,
     passedTurn = 2}
```

Enumeration to specify the current state of the game.

inactive: The game is not active. Execute ResetGame()-method to start it.

normalTurn: It is a player's turn to play a card

passedTurn: A turn was passed

PlayerType

```
public enum PlayerType{
     local = 0,
     AI = 1,
     custom = 2}
```

Enumeration to specify the type of a player.

local: Local player. The input is given through the LocalMove()-method.

AI: AI. The move is made automatically by the AI.

custom: Custom. The input is given through the CustomMove()-method. Can be used, for example, to input the moves of a remote player.

Variables

waitPeriod

```
public static int waitPeriod
```

time to wait between certain actions in milliseconds

Properties

seats

```
public static PlayerType[] seats
     { get;
     set; }
```

Number of players. From 2 to 6

Hands

```
public static List<Card>[] Hands
     { get;
     }
```

Lists of all players’ hand cards.

Table

```
public static List<Card> Table
     { get;
     }
```

List of cards on the table.

CurrentPlayer

```
public static int CurrentPlayer
     { get;
     }
```

The number of the current player. Ranges from 1 to length of ’seats’.

gameState

```
public static GameState gameState
     { get;
     }
```

The current state of the game. As a GameState enumeration.

Methods

ResetGame

```
public static void ResetGame()
```

Resets all variables related to the current game and stops the AI if needed.

StopAI

```
public static void StopAI()
```

This method can be used to stop the AI. This will abort the whole AI thread, meaning that the AI will not do a move. Should not be needed by the user.

LocalMove

```
public static void LocalMove(
     Card card)
```

Executes the given move if the current player is a local player. Passing is done by "playing" a heart of sevens.

card: The card used by the player.

CustomMove

```
public static void CustomMove(
     Card card)
```

Executes the given move if the current player is a custom player. Passing is done by "playing" a heart of sevens.

card: The card used by the player.

PossibleMoves

```
public static List<Card> PossibleMoves()
```

Finds all possible moves for the current player

Returns: List of all possible moves for the current player as Cards

PossibleMoves

```
public static List<Card> PossibleMoves(
     List<Card> handCards,
     List<Card> tableCards)
```

Finds all possible moves for the given hand cards and table

Returns: List of all possible moves for the given parameters as Cards

TopTrumps

C# TopTrumpsAI.cs

Namespaces

CGAIE

```
namespace CGAIE
```

Classes

TopTrumpsAI

```
public static class TopTrumpsAI
```

AI for the card game Top Trumps.

Enumerations

TopTrumpsAIType

```
public enum TopTrumpsAIType{
     random = 0,
     easy = 1,
     medium = 2,
     hard = 3,
     veryHard = 4}
```

Enumeration to define the difficulty of the AI.

Variables

AIType

```
public static TopTrumpsAIType AIType
```

The type of the AI. As a TopTrumpsAIType enumeration.

Methods

MakeMove

```
public static string MakeMove(
     CustomCard topCard,
     CustomDeck deck,
     List<CustomCard>[] unShuffledPiles)
```

The AI makes a move using the specified AI type.

topCard: The AI’s card that it is about to play.
deck: The CustomCard deck used in the game. The deck should be "full", containing all cards that are circulated in the game.
unShuffledPiles: Array of everyone’s cards that they have won but not yet shuffled back to their hand.

Returns: Top Trumps move as a string, corresponding the CustomCard attribute that will be played.

C# TopTrumpsEngine.cs

Namespaces

CGAIE

```
namespace CGAIE
```

Classes

TopTrumpsEngine

```
public static class TopTrumpsEngine
```

General Top Trumps engine. Rules:

The game can use any kind of cards. Not just classic playing cards.

Each card in the deck has same attributes, but with different values.

The deck is divided among all players as evenly as possible, and all players are only able to see the topmost card of their own cards.

First player initiates the first round, and from there, each round is initiated by the winner of the previous round.

The player initiating a round picks one of the attributes that the cards have, and then everyone plays their card on the table.

The player with the card that has the highest value in the picked attribute wins the round and gets the cards from other players.

In case of a tie, the cards go to the side instead and the next round is a tie-breaker, where the winner of that round gets all the cards from the previous round.

If a player does not have cards to take part in a tie-breaker, they lose automatically.

When a player runs out of cards from their hand, they shuffle the pile that they have accumulated through wins.

When a player is truly out of cards, they are out of the game.

The player who collects the entire deck, wins.

Enumerations

GameState

```
public enum GameState{
     inactive = 0,
     normalTurn = 1,
     normalTurnAfterTie = 2,
     endOfRound = 3}
```

Enumeration to specify the current state of the game.

inactive: The game is not active. Execute ResetGame()-method to start it.

normalTurn: It is a player’s turn to choose an attribute.

normalTurnAfterTie: It is a player’s turn to choose an attribute after a tie.

endOfRound: The game is between turns.

PlayerType

```
public enum PlayerType{
     local = 0,
     AI = 1,
     custom = 2}
```

Enumeration to specify the type of a player.

local: Local player. The input is given through the LocalMove()-method.

AI: AI. The move is made automatically by the AI.

custom: Custom. The input is given through the CustomMove()-method. Can be used, for example, to input the moves of a remote player.

Variables

NoTieBreakers

```
public static bool NoTieBreakers
```

Are there tie-breakers in Top Trumps. If not, after a tie, everyone takes their cards back and puts them on the "unshuffled" cards.

PrivateTieBreakers

```
public static bool PrivateTieBreakers
```

Should tie-breakers only include players who tied in the first place.

LowestValueBeatsHighest

```
public static bool LowestValueBeatsHighest
```

If the lowest and highest possible value of the relevant attribute are both played during the same round, does the lowest value player win instead. For example, a two (lowest value) would win against the ace (highest value).

deck

```
public static CustomDeck deck
```

The deck used for the game. This deck has to be defined before the game can start.

The deck consists of an array of attributes that each of its cards should have, as well as a list of the CustomCards themself.

waitPeriod

```
public static int waitPeriod
```

time to wait between rounds in milliseconds

Properties

seats

```
public static PlayerType[] seats
     { get;
     set; }
```

Number of players. From 2 to 6

Hands

```
public static List<CustomCard>[] Hands
     { get;
     }
```

List of cards for each player.

UnShuffledHands

```
public static List<CustomCard>[] UnShuffledHands
     { get;
     }
```

List of cards each player has won, waiting to be shuffled in their deck.

Table

```
public static List<CustomCard> Table
     { get;
     }
```

List of cards on the table. Everyone’s top-most card goes on the "table".

AdditionalCards

```
public static List<CustomCard> AdditionalCards
     { get;
     }
```

List of additional cards on the table, or the "pot". Cards from a tie end up here and the winner of the tie-breaker gets them.

CurrentPlayer

```
public static int CurrentPlayer
     { get;
     }
```

The current player’s number. Ranges from 1 to length of ’seats’.

gameState

```
public static GameState gameState
     { get;
     }
```

The current state of the game. As a GameState enumeration.

CurrentAttribute

```
public static string CurrentAttribute
     { get;
     }
```

The attribute currently being compared. Chosen by the previous round’s winner.

TieBreakerParticipants

```
public static List<int> TieBreakerParticipants
     { get;
     }
```

Players that take part in the current tie-breaker (only used if ’PrivateTieBreakers’ is True.

Methods

ResetGame

```
public static void ResetGame(
     bool warnIfIncompleteDeck = true)
```

Resets all variables related to the current game and stops the AI if needed.

StopAI

```
public static void StopAI()
```

This method can be used to stop the AI. This will abort the whole AI thread, meaning that the AI will not do a move. Should not be needed by the user.

LocalMove

```
public static void LocalMove(
     string attribute)
```

Executes the given move if the current player is a local player.

card: The used hand card.

CustomMove

```
public static void CustomMove(
     string attribute)
```

Executes the given move if the current player is a custom player.

card: The used hand card.

PossibleMoves

```
public static string[] PossibleMoves()
```

Finds all possible moves for the current player

Returns: List of all possible moves for the current player as Cards