OOP ASSIGNMENT - Go Fish!

# Introduction – How to play Go Fish (the programs rules)

Go fish has a small number of players each with a hand of 7 cards. One player asks another for all the cards they have of a certain rank. If they have those cards then the player must publically hand them over, if not the original player “goes fishing” and draws a card from the deck. When a player has 4 of a kind they place that card face up on the table, when all books have been won the player with the most books wins.

There’re rules that go Fish has at this program does not implement, such as you must have a card in a particular rank to request a card of that rank.

# Non-Delphi Source

Originally I wanted to make a networked card game. So that the users could play without seeing each other’s hands. So I developed some python to use sockets on a LAN, which was a valid method of networking on my old school network so I built this network handler class:

NetworkHandeler.py

1. **from** socket **import** \*
2. **import** time
4. port = 1234
6. **class** network:
7. host = ""
8. timeOut = 1
9. buffer = 1024
11. **def** \_\_init\_\_(self, host):
12. self.host = host
14. **def** Send(self, data):
15. addr = (self.host, port)
16. Sock = socket(AF\_INET, SOCK\_DGRAM)
17. Sock.sendto(data.encode("utf8", 'ignore'), addr)
18. Sock.close()
20. **def** SendData(self, data):
21. self.Send(data)
22. response = self.Listen()
23. **print**("Got response ", response)
24. **if** (response == "ACK"):
25. **return** "RECIVED ACK"
26. **else**:
27. **return** "NET FAIL"
29. **def** Listen(self):
30. addr = ("", port)
31. Sock = socket(AF\_INET, SOCK\_DGRAM)
32. Sock.bind(addr)
33. (data, hostAddr) = Sock.recvfrom(self.buffer)
34. data = data.decode("utf-8")
35. Sock.close()
36. **return** data
38. **def** ListenForData(self):
39. Data = self.Listen()
40. time.sleep(0.5)
41. self.Send("ACK")
42. **return** Data
44. **def** \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):
45. """some clean up code probably should exist"""

48. n = network("172.20.24.65")
49. n.ListenForData()

I then planned for this to communicate to Delphi over a CSV file system as I had already done the hard work of building a decent file handling unit in Delphi and I had split on deliminator capabilities:

CSVHandler.py

1. **import** csv
2. **import** time

5. **class** CSV:
6. filename = ""
7. MsgID = 0
9. **def** \_\_init\_\_(self, filename):
10. self.filename = filename
11. self.CSVAppend("PY IS ALIVE")
13. **def** CSVRead(self):
14. rows = []
15. with open(self.filename, 'r') as csvfile:
16. csvreader = csv.reader(csvfile)
17. **for** row **in** csvreader:
18. rows.append(row)
19. csvfile.close()
20. self.MsgID = rows[-2][0]
21. # Note this is -2 because of the double line spacing bug
22. **return** rows
24. **def** CSVAppend(self, data):
25. with open(self.filename, 'a') as csvfile:
26. csvwriter = csv.writer(csvfile)
27. **for** i **in** range(0, len(data)):
28. row = [self.MsgID, time.time(), data]
29. csvwriter.writerow(row)
30. self.MsgID += 1
31. csvfile.close()
33. **def** CSVClear(self):
34. csvfile = open(self.filename, "w+")
35. csvfile.close()
37. **def** \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):
38. self.CSVAppend("PY OBJ DESTROYED")

I then tried this on the schools network and promptly ran into fire wall problems…. So I started trying to get around the fire wall problems by building a node.js server and communicating over http.

1. **var** http = require('http');
2. **var** url = require('url');
3. **var** md5 = require('md5');
5. **function** User() {
6. **const** ID = md5(Math.random());
8. **this**.getUserID = **function** () {
9. **return** ID;
10. }
11. }
13. **function** Player(User, num) {
14. **var** number = num;
15. **var** hand = [];
16. **var** ranksWon = [];
18. **this**.addToHand = **function**(card) {
19. hand.push(card);
20. };
22. **this**.removeFromHand = **function** (card) {
23. **var** pos = hand.indexOf(card);
24. hand.splice(pos, 1);
25. };
27. **this**.getHand = **function** () {
28. **return** hand;
29. };
31. **this**.getID = **function** () {
32. **return** User.getUserID();
33. }
34. }
36. **function** Game(users) {
37. **var** deck = [];
38. **var** Finished = **false**;
40. **this**.players = users;
42. **this**.dealToPlayer = **function** (player) {
43. **for** (x = 0; x < 8; x++) {
44. **this**.players[player].addToHand(deck.pop());
45. }
46. };
48. **this**.getDataForUser = **function** (userID) {
49. **for** (i = 0; i < **this**.players.length; i++) {
50. **if** (userID == **this**.players[i].getUserID()) {
51. **var** player = **this**.players[i];
52. **break**
53. }
54. }
56. **return** "HAND: " + player.getHand().toString();
57. }

60. **for** (i = 0; i < 52; i++) {
61. deck[i] = i;
62. }
64. **for** (i = 0; i < numOfPlayers+1; i++) {
65. **this**.players[i] = **new** Player(i);
66. **this**.dealToPlayer(i);
67. }
69. }
71. **function** Lobby() {
72. **this**.waitingUsers = [];
73. **this**.users = [];
74. **this**.games = [];
76. **this**.maybeNewGame = **function** () {
77. **if** (**this**.waitingUsers.length > 1) {
79. **var** g = **new** Game(**this**.waitingUsers);
80. **this**.games.push(g);
81. **this**.waitingUsers.empty();
83. **return** **this**.games.indexOf(g);
84. } **else** {
85. **return** **false**
86. }
87. };
89. **this**.maybeRemoveGame = **function** () {
90. **for** (i = 0; i < **this**.games.length; i++) {
91. **if** (**this**.games[i].isFinished()) {
92. **this**.games.splice(i, 1);
93. }
94. }
95. };
96. }
98. //=========---Main Program---=========
100. lobby = **new** Lobby();
102. http.createServer(**function** (req, res) {
103. **var** q = url.parse(req.url, **true**);
104. **var** qdata = q.query;
106. **var** userID = qdata.userID;
107. **var** gameNum =  qdata.gameNum;
108. **var** userIndex = **null**;
110. **for**(**var** i = 0; i < lobby.users.length; i++) {
111. **if** (lobby.users[i].getUserID() == userID) {
112. userIndex = i;
113. **break**;
114. }
115. }
117. **if** (userIndex == **null**) {
118. **var** u = **new** User();
119. lobby.users.push(u);
120. userID = u.getUserID();
121. res.write("USERID: " + userID.toString());
122. lobby.waitingUsers.push(u);
123. } **else** {
124. res.write("USERID: " + userID + ";");
125. **var** u = lobby.users[userIndex];
126. **if** (!lobby.waitingUsers.includes(u)) { //Contains is not a function
127. lobby.waitingUsers.push(u);
128. } **else** {
129. res.write(lobby.games[gameNum].getDataForUser(userID));
130. }
131. }
133. **var** potentialGame = lobby.maybeNewGame();
135. **if** (potentialGame != **false**) {
136. res.write("INIT NEW GAME;")
137. **var** game = potentialGame;
138. **for** (i = 0; i < game.players.length; i++) {
139. **if** (userID == game.players.users[i].getUserID()) {
140. **var** user = game.players.users[i];
141. **break**
142. }
143. }
144. **var** playerNumber = game.players.indexOf(user);
145. res.write("PlayerNumber " + playerNumber.toString());
146. }
148. res.end();
150. }).listen(8080);

The program above does everything that it is meant to do, it’s just it needs to do a lot and I focused on the details that didn’t really matter the most first. So after Easter I started work with just making the single player version of the game.

I didn’t have time for a class diagram but in essence, program creates a server object which creates a lobby object which handles an array of user objects which can be placed taken out of the lobby and placed into a game object, theory being that you could be able to have many game objects running at once.

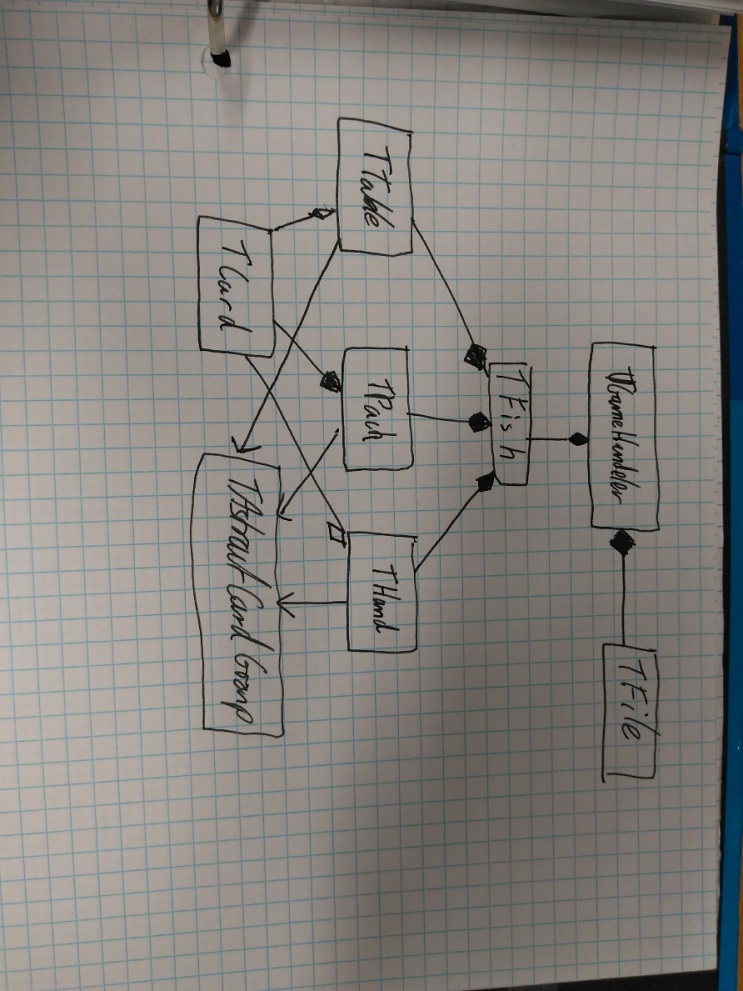
I did successfully manage to get this set up on a raspberry pi at home with port forwarding.

# The Delphi Source.

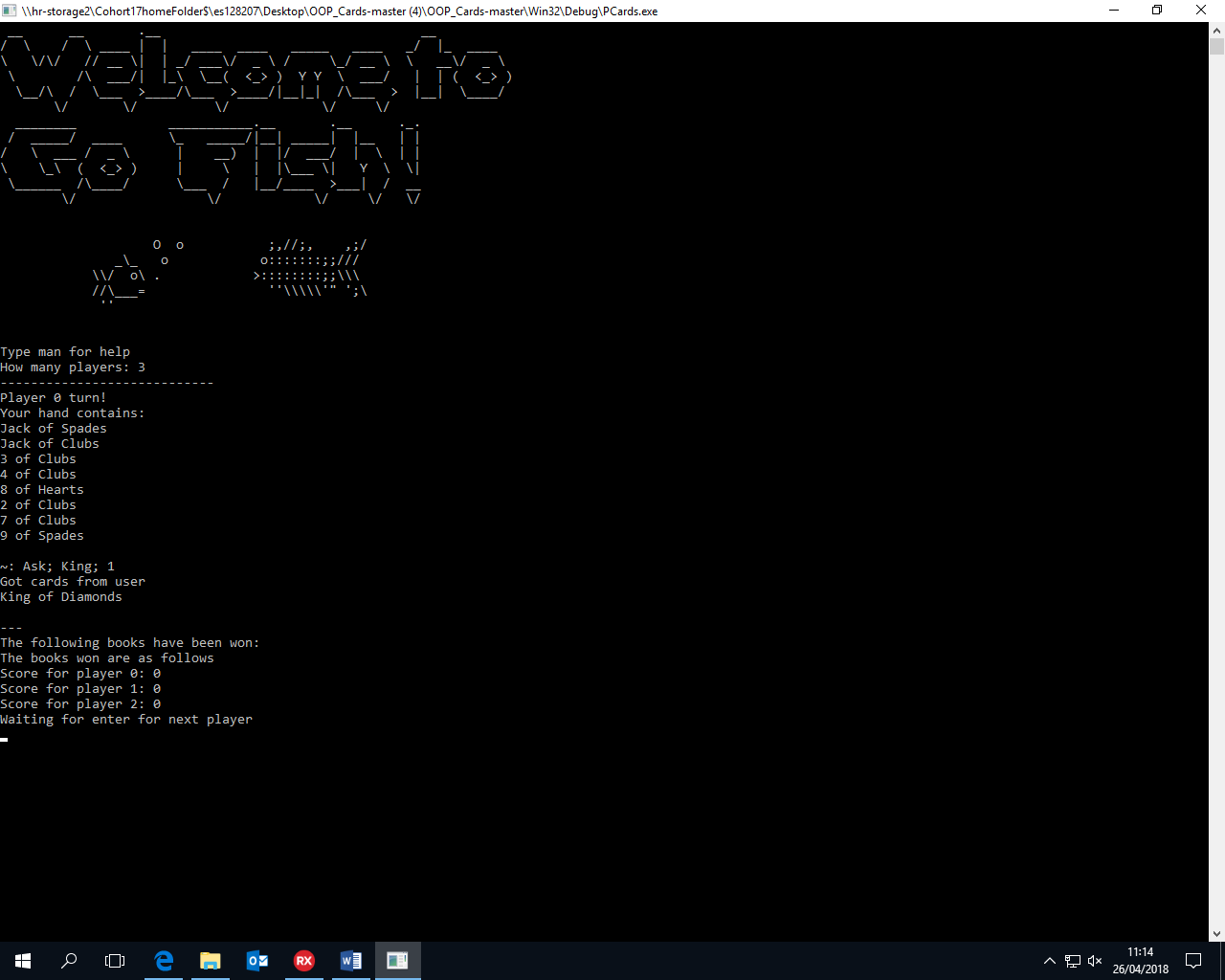
The game has many classes, all custom writen, here is a summary:

|  |  |
| --- | --- |
| Class | Purpose |
| GameHandler | This class handles the user and interaction with it. All UI is done through this class.  The idea was that this class could load, in different game objects and play those games, and save them for playing later. |
| Fish | This is game class with methods that are specific to GoFish, e.g. ask a user for a card of a specific rank. |
| Hand | This is the “players” it contains the cards they hold the ability to search through their hand etc. |
| Pack | Parent class of all of the cards. |
| Table | Where the players can place their books of cards |
| AbstractCardGroup | The abstract class that, hand, pack, table classes are based off |
| Cards | The objects |
| FileHandeler | Handles Files |

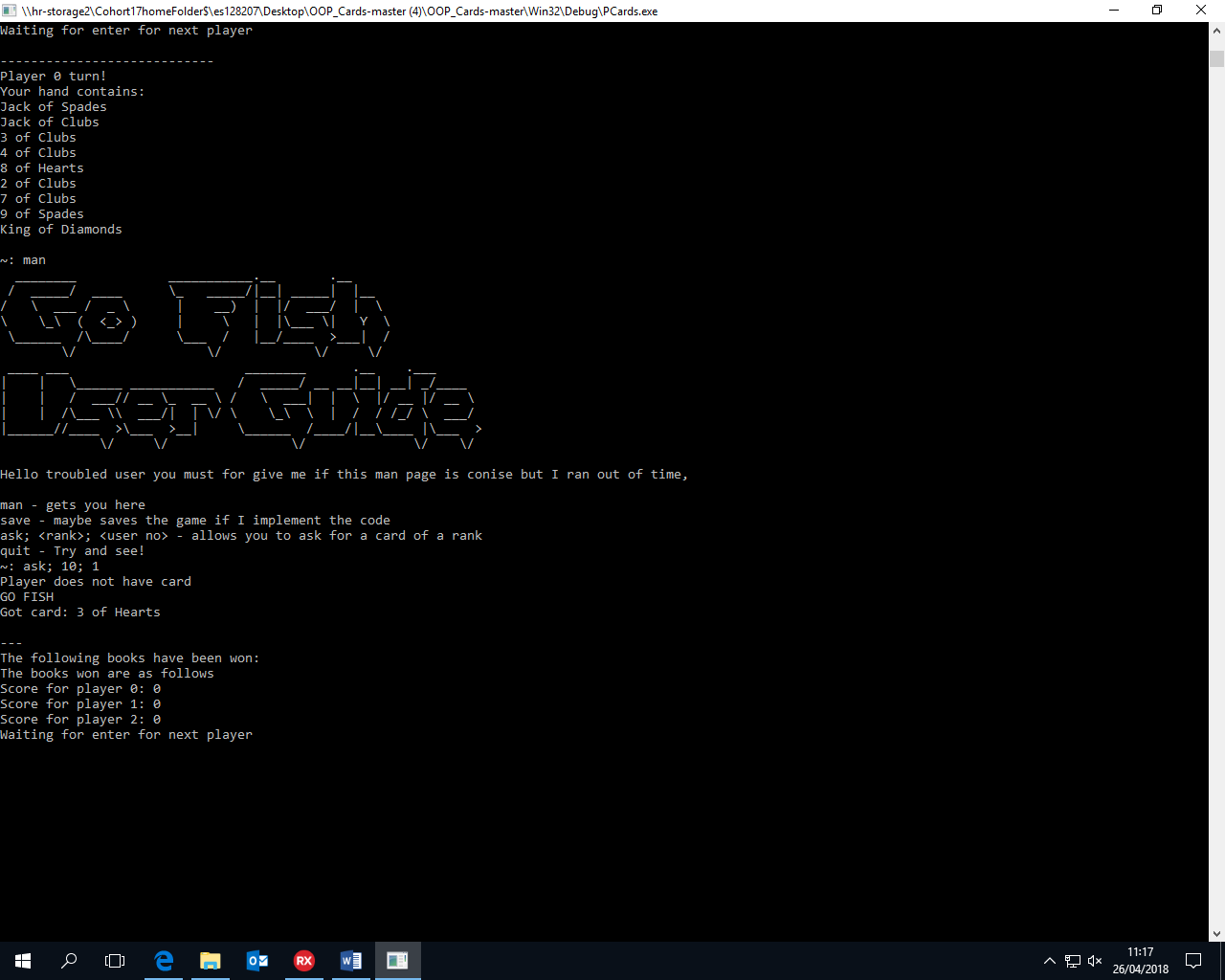
# My class digram:

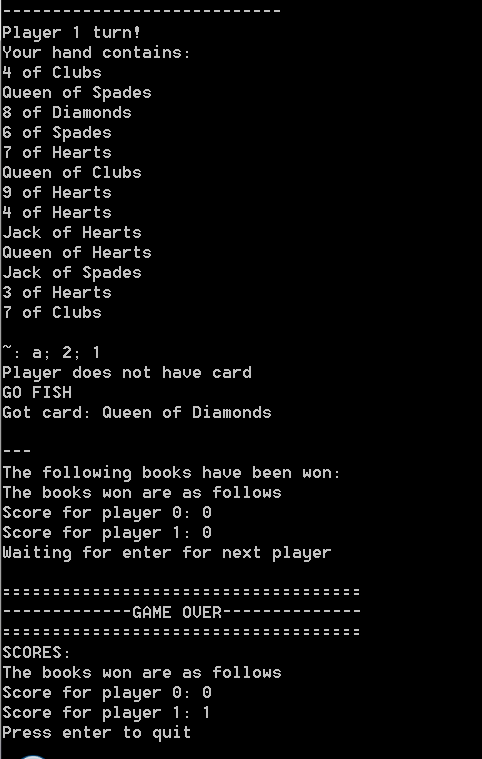


# Screen shots

Left is an example of my code running, player 0 asking for kings from player 1 and getting the king of Diamonds added to their hand.

It then displays the scores at the bottom.

Below is another user, confused about how to use the program. They got the man page and they asked for a card the users didn’t have and so were told GO FISH!

The screen shot left demonstrates the end condition, to have a total of 12 books won across players can be met. Because I didn’t feel like playing a game of go fish against myself, I up’d the initial value of TFish.totalScores up to 11 and then went fishing to get the last book. This is a fine thing to do as I know book winning works.

This screen shot does demonstrate a bug, the program will display the books for each player before ending the game at which point it will update the books and display the correct books for each player. As it eventually does the right thing I think that this is acceptable.

# TL;DR

I do not expect the reader to go through the ~1,000 lines of source appended to this document. So I would like to point out here short bits of code.

## Dictionaries

It is nicer for the user to address cards by an ID that they are familiar with e.g. “Ten of Hearts”, the problem with this is that Delphi only accepts ordinals in case statements which means that I would have to use the first letter of each card and “Two” and “Ten” would then evaluate to the same. My solution is to use a dictionary to covert between user input data, and ID’s. As I added this after creating program output the program going from ID’s to user readable data does not use dictionaries, which is inefficient.

I have three dictionaries, for commands, ranks and suits.

## File Handling

My program contains ASCII art which is significantly easier to create in WordPad than Rad Studio, I also at one point had the intent on trying to get the program to save the game the user was currently playing. Hence the separation between TGameHandler and TFish.

The file handling unit is the one originally created for the Encryption assignment but has been modified to be object orientated.

## Menu

As is standard with all my programs they use a delimiter powered terminal like menu. This implementation has been extensively improved from previous iterations and now uses 1 dynamic array type rather than declaring fixed arrays and lists left right and centre.

## Classes

I would like to note that I have **not** used any of the card or pack classes from sharepoint, I have tried to make them fairly general and as such comply with the ideology of OOP for reusable code.

# Some testing

Initialization

|  |  |  |
| --- | --- | --- |
| **Input** | **Excepted Outcome** | **Actual Outcome** |
| How many players <= -1 | Ask for how many players again | Ask for how many players again |
| How many players <= 10 | Ask for how many players again | Ask for how many players again |
| Asdfadsf | The program will crash | Crash with EInOutError.InvalidNumericInput |

Main game – Note: testing done with 2 players

|  |  |  |
| --- | --- | --- |
| **Input** | **Excepted Outcome** | **Actual Outcome** |
| ThisIsNotAValidINPUT | Crash | Crash with EListError.ItemNotFound |
| man | Bring up man page | Brings up man page |
| ask; ace; 1 | An ace if there is an ace in the other players hand else a go fish | Got the Ace of Spades |
| On player 1’s turn:  ask; ace; 1 | The program to look for a card in player 1’s hand and add it to player one, or to go fish | The program went fishing.  On second try the program did the expected outcome. |
| Quit | The program to quit | Memory access violation. |
| ask; ace; -1 | Memory access violation | Memory access violation |

# Evaluation & Lessons Learnt

I was too ambitious and prioritized the wrong stuff, eg building the network handlers before I built the actual game. This is a lesson that I will have to bear in mind when doing course work. This resulted in insufficient time for testing and a program with many unresolved bugs, for example, the program will crash if the user does not behave as I did not have the time for proper input sanitization. I will have to work on my project management.

However, despite this I am pleased with myself for actually having a program that worked by the end of it all. It compiles runs and allows for a full play through of a game of go fish. It is object orientated and I don’t think too much of a mess to not be able to work with it and improve it in the future. The test plan illustrates the main weakness of this program which is it crashes very easily if the user does not behave. Time spent in redundancy is never wasted.

# Possible Improvements

Here is a short list of things that I would like to work differently about this:

* The other users hands should be properly hidden from the other users
  + It would be nice to get the networking working
* Hands should auto sort by rank
* Saving game mechanics
* Stricter adherence to the rules of Go Fish

# 20 pages of undocumented code:

PCards:

1. **program** PCards;
3. {$APPTYPE CONSOLE}
4. {$R \*.res}
6. **uses**
7. System.SysUtils,
8. UCard **in** 'UCard.pas',
9. UPack **in** 'UPack.pas',
10. UHand **in** 'UHand.pas',
11. UExceptions **in** 'UExceptions.pas',
12. UFish **in** 'UFish.pas',
13. UAbstractCardGroup **in** 'UAbstractCardGroup.pas',
14. UGameHandeler **in** 'UGameHandeler.pas',
15. UFileHandeling **in** 'UFileHandeling.pas';
17. **var**
18. GameHandeler: TGameHandeler;
20. **begin**
21. randomize();
22. GameHandeler := TGameHandeler.create();
24. GameHandeler.play;
26. **writeln**('Press enter to quit');
27. readln;
29. GameHandeler.destroy;
30. exit;
31. **end**.

UCard:

1. **unit** UCard;
3. **interface**
5. **uses**
6. System.SysUtils;
8. **type**
9. Tcard = **class**
10. **private**
11. Rank, Suit: **integer**;
12. **public**
13. **constructor** create(r: **integer**; s: **integer**);
14. **constructor** createfromstr(r: **string**; s: **string**);
15. **function** GetRank: **integer**;
16. **function** GetExplicitRank: **string**;
17. **function** GetExplicitSuit: **string**;
18. **function** GetExplicitCard: **string**;
19. **function** GetSuit: **integer**;
20. **function** GetScore: **integer**;
22. **end**;
24. **implementation**
26. { Tcard }
27. **constructor** Tcard.create(r, s: **integer**);
28. **begin**
29. Suit := s;
30. Rank := r;
31. **end**;
33. **constructor** Tcard.createfromstr(r: **string**; s: **string**);
34. **begin**
35. **case** s[1] **of**
36. 'S':
37. Suit := 0;
38. 'H':
39. Suit := 1;
40. 'C':
41. Suit := 2;
42. 'D':
43. Suit := 3;
44. **else**
45. Suit := strtoint(s);
46. **end**;
48. **case** r[1] **of**
49. 'A':
50. Rank := 0;
51. 'J':
52. Rank := 10;
53. 'Q':
54. Rank := 11;
55. 'K':
56. Rank := 12;
57. **else**
58. Rank := strtoint(r);
59. **end**;
60. **end**;
62. **function** Tcard.GetExplicitCard: **string**;
63. **begin**
64. result := self.GetExplicitRank + ' of ' + self.GetExplicitSuit;
65. **end**;
67. **function** Tcard.GetExplicitRank: **string**;
68. **begin**
69. **case** self.Rank **of**
70. 0:
71. result := 'Ace';
72. 10:
73. result := 'Jack';
74. 11:
75. result := 'Queen';
76. 12:
77. result := 'King';
78. **else**
79. result := inttostr(self.Rank + 1);
80. **end**;
81. **end**;
83. **function** Tcard.GetExplicitSuit: **string**;
84. **begin**
85. **case** self.Suit **of**
86. 0:
87. result := 'Spades';
88. 1:
89. result := 'Hearts';
90. 2:
91. result := 'Clubs';
92. 3:
93. result := 'Diamonds';
94. **end**;
95. **end**;
97. **function** Tcard.GetRank: **integer**;
98. **begin**
99. result := self.Rank;
100. **end**;
102. **function** Tcard.GetScore: **integer**;
103. **begin**
104. result := (self.GetRank \* 4) + self.GetSuit;
105. **end**;
107. **function** Tcard.GetSuit: **integer**;
108. **begin**
109. result := self.Suit;
110. **end**;


114. **end**.

UAbstractCardGroup:

1. **unit** UAbstractCardGroup;
3. **interface**
4. **uses**
5. UCard;
7. **type**
8. TArrayOfString = Tarray<**string**>;
9. TCards = **array** **of** TCard;
11. TAbstractCardGroup = **class** abstract
12. **protected**
13. cards: TCards;
14. **public**
15. **function** GetSize: **integer**;
16. **function** findPos(rank, suit: **integer**): **integer**; Overload;
17. **function** findPos(card: tcard): **integer**; Overload;
18. **function** findCard(rank, suit: **integer**): tcard; Overload;
19. **function** findCard(card: tcard): tcard; Overload;
20. **procedure** addTo(card: tcard);
21. **end**;

24. **implementation**
26. { TAbstractCardGroup }
28. **procedure** TAbstractCardGroup.addTo(card: tcard);
29. **begin**
30. setlength(cards, length(cards) + 1);
31. cards[High(cards)] := card;
32. **end**;
34. **function** TAbstractCardGroup.findCard(card: tcard): tcard;
35. **var**
36. i: **integer**;
37. **begin**
38. **for** i := 0 **to** 51 **do**
39. **if** (cards[i].GetRank = card.GetRank) **and** (cards[i].GetSuit = card.GetSuit)
40. **then**
41. result := cards[i];
42. **end**;
44. **function** TAbstractCardGroup.findCard(rank, suit: **integer**): tcard;
45. **var**
46. i: **integer**;
47. **begin**
48. **for** i := 0 **to** 51 **do**
49. **if** (cards[i].GetRank = rank) **and** (cards[i].GetSuit = suit) **then**
50. result := cards[i];
51. **end**;
53. **function** TAbstractCardGroup.findPos(rank, suit: **integer**): **integer**;
54. **var**
55. i: **integer**;
56. **begin**
57. **for** i := 0 **to** 51 **do**
58. **if** (cards[i].GetRank = rank) **and** (cards[i].GetSuit = suit) **then**
59. result := i;
60. **end**;
62. **function** TAbstractCardGroup.findPos(card: tcard): **integer**;
63. **var**
64. i: **integer**;
65. **begin**
66. **for** i := 0 **to** length(cards) **do**
67. **if** (cards[i] = card) **then**
68. result := i;
69. **end**;
71. **function** TAbstractCardGroup.GetSize: **integer**;
72. **begin**
73. result := length(cards);
74. **end**;
76. **end**.

UPack:

1. **unit** UPack;
3. **interface**
5. **uses**
6. System.SysUtils,
7. UCard, UAbstractCardGroup, UExceptions, System.Generics.Collections,
8. System.StrUtils;
10. **type**
11. TPack = **class**(TAbstractCardGroup)
12. **private**
13. pack\_top: **integer**;
14. **public**
15. Debug: Boolean;
16. **constructor** create;
17. **function** draw: Tcard;
18. **procedure** shuffle;
20. **destructor** destroy;
21. **end**;
23. **implementation**
25. { Tpack }
27. **constructor** TPack.create();
28. **var**
29. I: **integer**;
30. **begin**
31. setlength(cards, 52);
33. **for** I := 0 **to** 51 **do**
34. cards[I] := Tcard.create(I **mod** 13, I **mod** 4);
36. pack\_top := 0;
37. **end**;
39. **destructor** TPack.destroy;
40. **var**
41. I: **integer**;
42. **begin**
43. **for** I := 0 **to** 51 **do**
44. **begin**
45. cards[I].Free;
46. **end**;
47. **end**;
49. **function** TPack.draw: Tcard; // This should be done using a stack
50. **begin**
51. result := cards[pack\_top];
52. **if** pack\_top < 52 **then**
53. **begin**
54. inc(pack\_top);
55. **end**
56. **else**
57. **begin**
58. EOutofCards;
59. **end**;
60. **end**;
62. **procedure** TPack.shuffle;
63. **var**
64. temp: Tcard;
65. random\_pos: **integer**;
66. I: **integer**;
67. **begin**
68. **for** I := 0 **to** 52 **do**
69. **begin**
70. random\_pos := I + random(52 - I); // like random range
71. temp := cards[random\_pos];
72. cards[random\_pos] := cards[I];
73. cards[I] := temp;
74. **end**;
75. pack\_top := 0;
76. **end**;
78. **end**.

UHand:

1. **unit** UHand;
3. **interface**
5. **uses**
6. UPack, UAbstractCardGroup, UCard, Generics.Collections;
8. **type**
9. Thand = **class** (TAbstractCardGroup)
10. **private**
11. **procedure** removeCard(card: tcard);
12. **function** GetHandSize: **integer**;
13. **public**
14. **constructor** Create;
15. **function** getcontents: TCards;
16. **function** placecard(index: **integer**): tcard; Overload;
17. **function** placecard(card: tcard): tcard; Overload;
18. **function** findCardByRank(rank: **integer**): tcard;
19. **function** howManyOfRank(rank: **integer**): **integer**;
20. **destructor** destroy;
22. **end**;
24. **implementation**
26. { Thand }
28. **function** Thand.howManyOfRank(rank: **integer**): **integer**;
29. **var**
30. i: **integer**;
31. **begin**
32. result := 0;
33. **for** I := 0 **to** length(cards)-1 **do** **begin**
34. **if** cards[i].GetRank = rank **then**
35. inc(result);
36. **end**;
37. **end**;
39. **constructor** Thand.Create;
40. **var**
41. i: **integer**;
42. **begin**
43. setlength(cards, 0);
44. **end**;
46. **destructor** Thand.destroy;
47. **begin**
49. **end**;

52. **function** Thand.findCardByRank(rank: **integer**): tcard;
53. **var**
54. i: **integer**;
55. **begin**
56. **for** I := 0 **to** length(cards)-1 **do** **begin**
57. **if** cards[i].GetRank = rank **then** **begin**
58. result := cards[i];
59. break;
60. **end**;
61. **end**;
62. **end**;

65. **function** Thand.getcontents: TCards;
66. **begin**
67. result := cards;
68. **end**;
70. **function** Thand.GetHandSize: **integer**;
71. **begin**
72. result := length(cards);
73. **end**;

76. **function** Thand.placecard(card: tcard): tcard;
77. **begin**
78. result := card;
79. self.removeCard(card);
80. **end**;
82. **procedure** Thand.removeCard(card: tcard);
83. **var**
84. i: **integer**;
85. **begin**
86. delete(cards, findPos(card), 1);
88. **end**;
90. **function** Thand.placecard(index: **integer**): tcard;
91. **begin**
92. result := cards[index];
93. delete(cards, index, 1);
94. **end**;
96. **end**.

UFish:

1. **unit** UFish;
3. **interface**
5. **uses**
6. UHand, UPack, UCard, UAbstractCardGroup, UExceptions;
8. **type**
10. TArrayOfHand = **array** **of** THand;
11. TArrayOfInterger = **array** **of** **integer**;
13. TTable = **class**(TAbstractCardGroup)
14. **private**
15. **constructor** create;
16. **end**;
18. TFish = **class**
19. **private**
20. Hands: TArrayOfHand;
21. Table: TTable;
22. Scores: TArrayOfInterger;
23. TotalScore: **integer**;
24. deck: TPack;
25. **public**
26. **constructor** create(numberOfPlayers: **integer**);
27. **function** checkForBook(playerNum: **integer**): **boolean**;
28. **function** AskForCard(rank: **integer**; Handfrom: THand; Handto: THand): TCards;
29. **function** GoFish(hand: THand): TCard;
31. **destructor** destroy;
33. **property** players: TArrayOfHand read Hands **write** Hands;
34. **property** pack: TPack read deck **write** deck;
35. **property** books: TArrayOfInterger read scores **write** scores;
36. **property** TotalBooks: **integer** read TotalScore **write** TotalScore;
38. **end**;
40. **implementation**
42. { Fish }
44. **function** TFish.AskForCard(rank: **integer**; Handfrom: THand; Handto: THand)
45. : TCards;
46. **var**
47. HandFromContents: TCards;
48. GotCardFromPlayer: **boolean**;
49. I, r: **integer**;
50. hf: TCard;
51. test: TCard;
52. **begin**
53. GotCardFromPlayer := **false**;
54. setlength(result, 0);
55. HandFromContents := Handfrom.getcontents;
56. i := 0;
57. **while** i < length(HandFromContents) **do**
58. **begin**
59. **if** (HandFromContents[I].getRank = rank) **then**
60. **begin**
61. setlength(result, length(result) + 1);
62. result[length(result) - 1] := HandFromContents[I];
63. test := Handfrom.placecard(I);
64. Handto.AddTo(test);
65. GotCardFromPlayer := **true**;
66. **end**;
67. inc(i);
68. **end**;
69. **end**;
71. **function** TFish.checkForBook(playerNum: **integer**): **boolean**;
72. **var**
73. I, j: **integer**;
74. hand: THand;
75. **begin**
76. **for** I := 0 **to** 12 **do**
77. **begin**
78. **if** (Hands[playerNum].howManyOfRank(I) = 4) **then**
79. **begin**
80. hand := Hands[playerNum];
81. inc(Scores[playerNum]);
82. inc(totalScore);
83. **for** j := 0 **to** 3 **do**
84. Table.AddTo(hand.placecard(hand.findCardByRank(I)));
85. **end**;
86. **end**;
87. **end**;
89. **constructor** TFish.create(numberOfPlayers: **integer**);
90. **var**
91. I, x: **integer**;
92. **begin**
93. **if** numberOfPlayers > 4 **then**
94. **begin**
95. ETooManyPlayers;
96. **end**
97. **else**
98. **begin**
100. Table := TTable.create;
102. deck := TPack.create();
103. deck.shuffle;
105. setlength(Hands, numberOfPlayers);
106. setlength(Scores, numberOfPlayers);
108. **for** I := 0 **to** numberOfPlayers **do**
109. **begin**
110. scores[i] := 0;
111. Hands[I] := THand.create;
112. **for** x := 0 **to** 7 **do**
113. Hands[I].AddTo(deck.draw);
114. **end**;
115. **end**;
117. **end**;
119. **destructor** TFish.destroy;
120. **begin**
122. **end**;
124. **function** TFish.GoFish(hand: THand): TCard;
125. **var**
126. Card: TCard;
127. **begin**
128. Card := deck.draw;
129. hand.AddTo(Card);
130. result := Card;
131. **end**;
133. { TTable }
135. **constructor** TTable.create;
136. **begin**
137. setlength(cards, 0);
138. **end**;
140. **end**.

UGameHandeler.

1. **unit** UGameHandeler;
3. **interface**
5. **uses**
6. System.SysUtils, classes, UCard, UFileHandeling, UFish,
7. System.Generics.Collections, UAbstractCardGroup, math;
9. **type**
10. TArrayOfString = Tarray<**string**>;
11. TCards = **array** **of** TCard;
13. TGameHandeler = **class**
14. **private**
15. Delimiter: **string**;
17. commands: TDictionary<**string**, **integer**>;
18. RankNames: TDictionary<**string**, **integer**>;
19. SuitNames: TDictionary<**string**, **integer**>;
21. game: TFish;
22. currentUser: **integer**;
24. **public**
25. **constructor** create;
26. **function** GetUserIn: TArrayOfString;
27. **procedure** man;
28. **procedure** ExecuteUserInstruction(cmd: TArrayOfString);
29. **procedure** Welcome;
30. **procedure** play;
31. **function** interpretCard(input: **string**): TCard;
32. **function** interpretRank(input: **string**): **integer**;
33. **procedure** displayBooks;
34. Destructor destroy;
35. **end**;
37. **const**
38. manFilePath = 'man.txt';
40. **implementation**
42. { TGameHandeler }
44. **function** TGameHandeler.GetUserIn: TArrayOfString;
45. **var**
46. userin: **string**;
47. **begin**
48. **while** length(userin) < 3 **do**
49. **begin**
50. **write**('~: ');
51. readln(userin);
52. **end**;
54. result := userin.Split([Delimiter]);
56. **end**;
58. **function** TGameHandeler.interpretCard(input: **string**): TCard;
59. **var**
60. SplitInput: Tarray<**string**>;
61. rank, suit: **integer**;
63. **begin**
64. SplitInput := input.Split([' ']);
65. rank := RankNames.Items[SplitInput[0]];
66. suit := SuitNames.Items[SplitInput[2]];
67. result := game.pack.findCard(rank, suit);
68. **end**;
70. **function** TGameHandeler.interpretRank(input: **string**): **integer**;
71. **begin**
72. result := RankNames.Items[input];
73. **end**;
75. **constructor** TGameHandeler.create;
76. **var**
77. playerNumber: **integer**;
78. **begin**
79. Delimiter := '; ';
81. RankNames := TDictionary<**string**, **integer**>.create();
83. RankNames.add('Ace', 0);
84. RankNames.add('ace', 0);
85. RankNames.add('Two', 1);
86. RankNames.add('two', 1);
87. RankNames.add('2', 1);
88. RankNames.add('Three', 2);
89. RankNames.add('three', 2);
90. RankNames.add('3', 2);
91. RankNames.add('Four', 3);
92. RankNames.add('four', 3);
93. RankNames.add('4', 3);
94. RankNames.add('Five', 4);
95. RankNames.add('five', 4);
96. RankNames.add('5', 4);
97. RankNames.add('Six', 5);
98. RankNames.add('six', 5);
99. RankNames.add('6', 5);
100. RankNames.add('Seven', 6);
101. RankNames.add('seven', 6);
102. RankNames.add('7', 6);
103. RankNames.add('Eight', 7);
104. RankNames.add('eight', 7);
105. RankNames.add('8', 7);
106. RankNames.add('Nine', 8);
107. RankNames.add('nine', 8);
108. RankNames.add('9', 8);
109. RankNames.add('Ten', 9);
110. RankNames.add('ten', 9);
111. RankNames.add('10', 9);
112. RankNames.add('Jack', 10);
113. RankNames.add('jack', 10);
114. RankNames.add('Queen', 11);
115. RankNames.add('queen', 11);
116. RankNames.add('King', 12);
117. RankNames.add('king', 12);
119. SuitNames := TDictionary<**string**, **integer**>.create();
121. SuitNames.add('Spades', 0);
122. SuitNames.add('spades', 0);
123. SuitNames.add('Hearts', 1);
124. SuitNames.add('hearts', 1);
125. SuitNames.add('Clubs', 2);
126. SuitNames.add('clubs', 2);
127. SuitNames.add('Diamonds', 3);
128. SuitNames.add('diamonds', 3);
130. commands := TDictionary<**string**, **integer**>.create();
132. commands.add('m', 0);
133. commands.add('man', 0);
134. commands.add('s', 1);
135. commands.add('save', 1);
136. commands.add('a', 2);
137. commands.add('ask', 2);
138. commands.add('A', 2);
139. commands.add('Ask', 2);
140. commands.add('q', 3);
141. commands.add('quit', 3);
143. Welcome;
145. **repeat**
146. **write**('How many players: ');
147. readln(playerNumber);
148. playerNumber := playerNumber - 1 // -= !!!!!
149. **until** (playerNumber > 0) **and** (playerNumber < 5);
151. game := TFish.create(playerNumber);
153. **end**;
155. **destructor** TGameHandeler.destroy;
156. **begin**
157. RankNames.destroy;
158. SuitNames.destroy;
159. commands.destroy;
160. game.destroy;
161. **end**;
163. **procedure** TGameHandeler.ExecuteUserInstruction(cmd: TArrayOfString);
164. **var**
165. cards: UAbstractCardGroup.TCards;
166. i: **integer**;
167. **begin**
168. **case** commands.Items[cmd[0]] **of**
169. 0:
170. **begin**
171. man;
172. ExecuteUserInstruction(GetUserIn);
173. **end**;
174. // 1:
175. // Add code to save game
177. 2:
178. **begin**
180. cards := game.AskForCard(interpretRank(cmd[1]),
181. game.players[strToint(cmd[2])], game.players[currentUser]);
182. **if** length(cards) = 0 **then**
183. **begin**
184. **writeln**('Player does not have card');
185. **writeln**('GO FISH');
186. **writeln**('Got card: ', game.GoFish(game.players[currentUser])
187. .GetExplicitCard);
189. **writeln**('');
190. **writeln**('---');
191. **writeln**('The following books have been won:');
192. displayBooks;
193. **end**
194. **else**
195. **begin**
196. **writeln**('Got cards from user');
197. **for** i := 0 **to** length(cards) - 1 **do**
198. **writeln**(cards[i].GetExplicitCard);
200. **writeln**('');
201. **writeln**('---');
202. **writeln**('The following books have been won:');
203. displayBooks;
204. **end**;
206. **end**;
207. 3:
208. game.destroy;
209. **end**;
211. **end**;
213. **procedure** TGameHandeler.displayBooks;
214. **var**
215. i: **integer**;
216. **begin**
217. **writeln**('The books won are as follows');
218. **for** i := 0 **to** length(game.players) **do**
219. **begin**
220. **writeln**('Score for player ', inttoStr(i), ': ', inttoStr(game.books[i]));
221. **end**;
222. **end**;
224. **procedure** TGameHandeler.man;
225. **var**
226. manFile: TFile;
227. mancontents: TArrayOfString;
229. **begin**
230. manFile := TFile.create(manFilePath);
231. manFile.printfile;
232. manFile.destroy;
233. **end**;
235. **procedure** TGameHandeler.play;
236. **var**
237. i, x: **integer**;
238. cardsInHand: UAbstractCardGroup.TCards;
239. totalBooksWon: **integer**;
241. **begin**
243. game.TotalBooks := 0;
245. **while** 12 > game.TotalBooks **do**
246. **begin**
248. **for** i := 0 **to** length(game.players) **do**
249. **begin**
250. currentuser := i;
252. **writeln**('----------------------------');
253. **writeln**('Player ', inttoStr(i), ' turn!');
255. **writeln**('Your hand contains: ');
257. cardsInHand := game.players[i].getcontents;
259. **for** x := 0 **to** length(cardsInHand) - 1 **do**
260. **writeln**(cardsInHand[x].GetExplicitCard);
262. **writeln**('');
263. ExecuteUserInstruction(GetUserIn);
265. **if** game.checkForBook(currentUser) **then**
266. **begin**
267. **writeln**('YOU GOT A BOOK!');
268. **writeln**('The scores are as follows');
269. displayBooks;
270. **end**;
272. **writeln**('Waiting for enter for next player');
273. readln;
275. **end**;
276. **end**;
278. **writeln**('====================================');
279. **writeln**('-------------GAME OVER--------------');
280. **writeln**('====================================');
282. **writeln**('SCORES:');
283. displayBooks;
284. **end**;
286. **procedure** TGameHandeler.Welcome;
287. **const**
288. welcomeFilePath = 'ASCIIFISH.txt';
289. **var**
290. welcomeFile: TFile;
291. **begin**
292. welcomeFile := TFile.create(welcomeFilePath);
293. welcomeFile.printfile;
294. welcomeFile.destroy;
295. **end**;
297. **end**.
298. **unit** UFileHandeling;
300. **interface**
302. **uses**
303. sysutils;
305. **type**
306. TArrayOfString = Tarray<**string**>;
308. TFile = **class**
309. **public**
310. **constructor** Create(filename: **string**);
311. **function** getfilelength: **integer**;
312. **function** readfile: TArrayOfString;
313. **procedure** writefile(data: **string**);
314. **procedure** appendfile(data: **string**);
315. **procedure** printfile;
316. Destructor Destroy;
317. **private**
318. filename: **string**;
319. **end**;
321. **implementation**
323. { TFile }
325. **procedure** TFile.appendfile(data: **string**);
326. **var**
327. myfile: textfile;
328. **begin**
329. assignfile(myfile, filename);
330. append(myfile);
331. **writeln**(myfile, data);
332. closefile(myfile);
333. **end**;
335. **constructor** TFile.Create(filename: **string**);
336. **begin**
337. self.filename := filename;
338. **end**;
340. **destructor** TFile.Destroy;
341. **begin**
343. **end**;
345. **function** TFile.getfilelength(): **integer**;
346. **var**
347. myfile: textfile;
348. count: **integer**;
350. **begin**
351. assignfile(myfile, self.filename);
352. reset(myfile);
353. **while** **not** eof(myfile) **do**
354. **begin**
355. readln(myfile, count);
356. count := count + 1;
357. **end**;
358. result := count;
359. **end**;

362. **procedure** TFile.printfile();
363. **var**
364. filecontents: TArrayOfString;
365. i: **integer**;
366. **begin**
367. filecontents := self.readfile();
368. **for** i := 0 **to** length(filecontents) - 1 **do**
369. **begin**
370. **writeln**(filecontents[i])
371. **end**;
372. **end**;
374. **function** TFile.readfile(): TArrayOfString;
375. **var**
376. myfile: textfile;
377. line, contents: **string**;
378. count: **integer**;
379. i: **integer**;
381. **begin**
382. count := 0;
383. assignfile(myfile, self.filename);
384. reset(myfile);
385. **while** **not** eof(myfile) **do**
386. **begin**
387. readln(myfile, line);
388. count := count + 1;
389. **end**;
391. setlength(result, count);
393. reset(myfile);
394. **for** i := 0 **to** count - 1 **do**
395. **begin**
396. readln(myfile, line);
397. result[i] := line;
398. **end**;
400. closefile(myfile);
401. **end**;
403. **procedure** TFile.writefile(data: **string**);
404. **var**
405. myfile: textfile;
406. **begin**
407. assignfile(myfile, self.filename);
408. rewrite(myfile);
409. **write**(myfile, data);
410. closefile(myfile);
411. **end**;
413. **end**.

UExceptions:

1. **unit** UExceptions;
3. **interface**
5. **uses**
6. System.SysUtils;
8. **procedure** EOutOfCards;
9. **procedure** ETooManyPlayers;
11. **implementation**
13. **procedure** EOutOfCards;
14. **begin**
15. **writeln**('Insufficent cards, can not draw requested card');
16. **raise** Exception.Create('Out of cards');
17. **end**;
19. **procedure** ETooManyPlayers;
20. **begin**
21. **writeln**('Too many players');
22. **raise** Exception.Create('Too many players');
23. **end**;
25. **end**.