----------cards.py----------

```
import random

class Card(object):
    """ A card object with a suit, rank, and file name.
    The file name refers to the card's image on disk."""

    RANKS = tuple(range(1, 14))
    SUITS = ('Spades', 'Hearts', 'Diamonds', 'Clubs')
    BACK_NAME = 'DECK/b.gif'

    def __init__(self, rank, suit):
        """Creates a card with the given rank and suit."""
        if not (rank in Card.RANKS):
            raise RuntimeError('Rank must be in ' + str(Card.RANKS))
        if not (suit in Card.SUITS):
            raise RuntimeError('Suit must be in ' + str(Card.SUITS))
        self.rank = rank
        self.suit = suit
        self.fileName = 'DECK/' + str(rank) + suit[0].lower() + '.gif'

    def __str__(self):
        """Returns the string representation of a card."""
        if self.rank == 1:
            rank = 'Ace'
        elif self.rank == 11:
            rank = 'Jack'
        elif self.rank == 12:
            rank = 'Queen'
        elif self.rank == 13:
            rank = 'King'
        else:
            rank = self.rank
        return str(rank) + ' of ' + self.suit


class Deck(object):
    def __init__(self):
        self.cards = []
        for suit in Card.SUITS:
            for rank in Card.RANKS:
                self.cards.append(Card(rank, suit))

    def __len__(self):
        return len(self.cards)

    def isEmpty(self):
        return len(self.cards) == 0

    def deal(self):
        return self.cards.pop()

    def shuffle(self):
        random.shuffle(self.cards)

def testCards():
    card = Card(13, "Hearts")
    print("Pass: Executed as expected with no error:", card)

    try:
        card = Card(13, "Heart")
    except RuntimeError as exc:
        print("Pass: Expected runtime error because used incorrect suit")

if __name__ == "__main__":
    testCards()
```

----------connectFour.py----------

```
from game import Game

class ConnectFour(Game):
   def __init__(self):
      super().__init__()
      self.board = [[' ' for x in range(7)] for y in range(6)]
      self.tokens = ["R", "B"]

   def getTurn(self):
      # Returns the token of the current turn
      return self.tokens[self.turn % len(self.tokens)]

   def step(self, column):

      # Obtain the row it ended up
      row = self.placeToken(column, self.tokens[self.turn % len(self.tokens)])

      # Only perform move if possible
      if row != -1:
         # Check for winning move
         self.gameOver = self.winningMove(column, row)

         super().step()

   def winningMove(self, column, row):
      return self.checkHorizontal(column, row) or \
             self.checkVertical(column, row) or \
             self.checkDiagonals(column, row)

   def checkHorizontal(self, column, row):
      # Check left and right
      tokenCount = 0
      for i in range(column, len(self.board[row])):
         if self.board[row][column] == self.board[row][i]:
            tokenCount += 1
         else:
            break

      if tokenCount < 4:
         for i in range(column - 1, -1, -1):
            if self.board[row][column] == self.board[row][i]:
               tokenCount += 1
            else:
               break

      return tokenCount >= 4

   def checkVertical(self, column, row):
      # Check down from position until we go off board or hit a color that is not ours
      tokenCount = 0
      for i in range(row, len(self.board)):
         if self.board[row][column] == self.board[i][column]:
            tokenCount += 1
         else:
            break

      return tokenCount >= 4

   def checkDiagonals(self, column, row):
      return self.genericDiagonalCheck(column, row, 1, 1) or \
             self.genericDiagonalCheck(column, row, -1, 1)

   def genericDiagonalCheck(self, column, row, colMod, rowMod):
      tokenCount = 0
      r, c = row, column
```

```
      while r < len(self.board) and c < len(self.board[row]) and r >= 0 and c >= 0:
         if self.board[r][c] == self.board[row][column]:
            tokenCount += 1
            r += rowMod
            c += colMod
         else:
            break

      if tokenCount < 4:
         r, c = row - rowMod, column - colMod
         while r < len(self.board) and c < len(self.board[row]) and r >= 0 and c >= 0:
            if self.board[r][c] == self.board[row][column]:
               tokenCount += 1
               r -= rowMod
               c -= colModj

            else:
               break

      return tokenCount >= 4

   def getInput(self):
      user_input = input("Enter the column for turn " + self.getTurn() + ": ")

      if user_input.isdigit():
         value = int(user_input)
         if value < 0 or value > 6:
            print("Value is out of range.  Try again!")
            return self.getInput()
         else:
            return value
      else:
         print("Incorrect input.  Try again!")
         return self.getInput()


   def placeToken(self, column, token):
      # place a token in column move for turn
      # start at bottom and work way up

      row = len(self.board) - 1

      while(self.board[row][column] != ' '):
         row -= 1

         # Can't place any more tokens in this column
         if row < 0:
            return -1

      self.board[row][column] = token

      return row


   def __str__(self):
      display = "\n".join(["| " + " | ".join(x) + " |" for x in self.board])

      display += "\n"
      display += "-----" * len(self.board) + "\n"

      display += " " + " ".join([" " + str(x) + " "  for x in range(len(self.board[0]))]) +
"\n"

      return display
```

```
def main():
   game = ConnectFour()
   game.main()
   game.reset()
   game.main()


if __name__ == '__main__':
   main()


-------------------------game.py-------------------
from abc import ABC, abstractmethod

class Game(ABC):

   def __init__(self):
      self.turn = 0
      self.gameOver = False

   def isGameOver(self):
      return self.gameOver

   @abstractmethod
   def step(self):
      if not self.gameOver:
         self.turn += 1

   def getTurn(self):
      return self.tokens[self.turn % len(self.tokens)]

   @abstractmethod
   def getInput(self):
      pass

   @abstractmethod
   def __str__(self):
      pass

   def main(self):
      while not self.isGameOver():
         print(self)

         self.step(self.getInput())

      print("\nGame Over!\n")
      print(self)

   def reset(self):
      type(self).__init__(self)



-------------------------ticTacToe.py-------------------
from game import Game

class TicTacToe(Game):
   def __init__(self):
      super().__init__()
      self.board = [[" " for x in range(3)] for y in range(3)]
      self.tokens = ["X", "O"]

   def getTurn(self):
      return self.tokens[self.turn % 2]
```

```
   def isGameOver(self):
      return self.gameOver

   def getInput(self):
      row, column = input("Enter row and column separated by a space: ").split()

      return (int(row), int(column))

   def step(self, move):

      row, column = move
      self.board[row][column] = self.tokens[self.turn % 2]

      # Manually check all possible wins
      if self.board[0][0] != " " and \
              self.board[0][0] == self.board[0][1] and \
              self.board[0][1] == self.board[0][2]:
         self.gameOver = True
      elif self.board[0][0] != " " and \
                self.board[0][0] == self.board[1][0] and \
                self.board[1][0] == self.board[2][0]:
         self.gameOver = True
      elif self.board[0][0] != " " and \
                self.board[0][0] == self.board[1][1] and \
                self.board[1][1] == self.board[2][2]:
         self.gameOver = True
      elif self.board[2][0] != " " and \
                self.board[2][0] == self.board[1][1] and \
                self.board[1][1] == self.board[0][2]:
         self.gameOver = True
      elif self.board[0][1] != " " and \
                self.board[0][1] == self.board[1][1] and \
                self.board[1][1] == self.board[2][1]:
         self.gameOver = True
      elif self.board[0][2] != " " and \
                self.board[0][2] == self.board[1][2] and \
                self.board[1][2] == self.board[2][2]:
         self.gameOver = True
      elif self.board[1][0] != " " and \
                self.board[1][0] == self.board[1][1] and \
                self.board[1][1] == self.board[1][2]:
         self.gameOver = True

      super().step()

   def __str__(self):
       return "\n-+-+-\n".join(["|".join(row) for row in self.board]) + "\n"


if __name__ == "__main__":
   game = TicTacToe()
   game.main()
-------------------------warGame.py-------------------
from cards import Deck
from game import Game

class WarGame(Game):
   """Plays the game of War."""

   def __init__(self):
      """Sets up the two players, the war pile, the deck, and the
      game state."""
      super().__init__()
      self.player1 = Player()
      self.player2 = Player()
      self.warPile = []
      self.gameState = ""
      self.deck = Deck()
      self.deck.shuffle()
      self.deal()

   def __str__(self):
      """Returns the game state."""
      if self.isGameOver():
         return self.winner()

      return self.gameState

   def deal(self):
      """Deals 26 cards to each player."""
      while not self.deck.isEmpty():
          self.player1.addToUnplayedPile(self.deck.deal())
          self.player2.addToUnplayedPile(self.deck.deal())

   def getInput(self):
      pass

   def step(self, dummyInput):
      """Makes one move in the game, and returns the two cards
      played."""

      super().step()

      card1 = self.player1.getCard()
      card2 = self.player2.getCard()
      self.warPile.append(card1)
      self.warPile.append(card2)
      self.gameState = "Player 1: " + str(card1) + "\n" +\
                       "Player 2: " + str(card2)
      if card1.rank == card2.rank:
         self.gameState += "\nCards added to War pile\n"
      elif card1.rank > card2.rank:
         self.transferCards(self.player1)
         self.gameState += "\nCards go to Player 1\n"
      else:
         self.transferCards(self.player2)
         self.gameState += "\nCards go to Player 2\n"

      self.gameOver = self.player1.isDone() or self.player2.isDone()

   def transferCards(self, player):
      """Transfers cards from the war pile to the player's
      winnings pile."""
      while len(self.warPile) > 0:
         player.addToWinningsPile(self.warPile.pop())


   def winner(self):
      """Returns None if there is no winner yet.  Otherwise,
      returns a string indicating the player who won with each
      player's number of cards, or a tie."""
      if self.player1.isDone() or self.player2.isDone():
         count1 = self.player1.winningsCount()
         count2 = self.player2.winningsCount()
         if count1 > count2:
            return "Player 1 wins, " + str(count1) + " to " +\
                   str(count2) +"!"

         elif count2 > count1:
            return "Player 2 wins, " + str(count2) + " to " +\
                   str(count1) +"!"
         else:
```

```
            return "The game ends in a tie!\n"
      else:
         return None

class Player(object):
   """Represents a War game player."""

   def __init__(self):
      """Sets up the player's unplayed and winnings piles."""
      self.unplayed = []
      self.winning = []

   def __str__(self):
      """Returns a description of the player's winnings pile."""
      winPile = "Winning pile:\n"
      for card in self.winning:
         winPile += str(card) + "\n"
      return winPile

   def addToUnplayedPile(self, card):
      """Adds card to the player's unplayed pile."""
      self.unplayed.append(card)

   def addToWinningsPile(self, card):
      """Adds card to the player's winnings pile."""
      self.winning.append(card)

   def getCard(self):
      """Removes and returns a card from the player's unplayed pile."""
      return self.unplayed.pop()

   def isDone(self):
      """Returns True if the player's unplayed pile is empty,
      or False otherwise."""
      return len(self.unplayed) == 0

   def winningsCount(self):
      """Returns the number of cards in the player's winnings pile."""
      return len(self.winning)


if __name__ == "__main__":
   game = WarGame()
   game.main()
```