# Gra Blackjack

Aby docenić możliwości programowania obiektowego trzeba zobaczyć grupę obiektów, które ze sobą współpracują, dlatego na zakończenie kursu stworzymy uproszczoną wersję gry karcianej blackjack.

## Zasady gry:

Graczom są rozdawane karty o wartościach punktowych. Każdy gracz stara się uzyskać sumę punktów najbliższą 21 tak **aby jej nie przekroczyć**.

Karty numerowane (**2 – 10**) są liczone zgodnie z wartością numeryczną.

**As** ma wartość **1** albo **11** (w zależności co jest dla gracza korzystniejsze), a każdy **walet, dama i król** dają wartość **10** punktów. Grać może maksymalnie 7 graczy.

## Opis projektu

Na początku komputer (krupier) rozdaje wszystkim graczom po dwie karty (włącznie z sobą). Gracze widzą wszystkie swoje karty, a komputer wyświetla ich sumę. Jedna z kart komputera pozostaje tymczasowo ukryta.

Następnie pierwszy z gracz może dobrać dodatkowe karty.

Każdy gracz może jednorazowo dobrać jedną kartę i powtarzać dobieranie wielokrotnie, sprawdzając swój wynik. Jeżeli gracz po dobraniu kart ma więcej niż 21 punktów, przegrywa tzw. furą.

Komputer musi dobrać karty dodatkowe dopóki ma mniej niż 17 punktów. Jeśli komputer dostanie furę, wszyscy gracze, którzy sami jej nie dostali, zostają zwycięzcami.

Na końcu gry suma punktów graczy, którzy nie przekroczyli 21 punktów jest porównywana z sumą uzyskaną przez komputer.  Wygrywa ten z uczestników gry, który ma sumę punktów bliższą lub równą 21.

# Karta

Zacznijmy stworzenia klasy **Card**, która będzie reprezentować karty do gry. Każda karta składa się z koloru (*ang. suit*) oraz swojej rangi (*ang. rank*).

W klasie Card umieścimy również stałe, które będą zawierać informacje o wszystkich możliwych kolorach i rangach dostępnych dla wystąpień klasy Card - obiektów.

```
class Card
   SUITS = ["♥", "♦", "♣", "♠"]
   RANKS = ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J",
"Q", "K"]
    def initialize(rank, suit)
       @rank = rank
       @suit = suit
   end
    def to_s
       @rank + @suit
   end
end
```

Przy tworzeniu obiektu konstruktor oczekuje 2 podstawowych informacji o karcie do zainicjalizowania obiektu.

```
card = Card.new("Q", "♣")
puts card.to_s
```

Obiekt z atrybutem **`rank`** o wartości “Q” i atrybutem **`suit`** o wartości “♣” reprezentuje damę trefl.

Metoda **`to_s`** wyświetli nam obiekt jako tekst, konkatenację atrybutu rank i suit.

# Ręka

Każdy gracz trzyma swoją kolekcję kart w dłoni. Początkowo ręka gracza jest pusta, możemy jednak dobrać karty - dodać je do naszej kolekcji. Klasa **Hand** będzie wyglądać następująco:

```
class Hand
   def initialize
       @cards = []
   end

   def add(card)
       @cards.append(card)
   end
end
```

Nowy obiekt klasy Hand zawiera atrybut **cards**, który jest listą innych obiektów (będziemy używać klasy Card).

Ponadto dodamy metodę **delete** usuwającą wszystkie karty gracza, **give** pozwalającą oddać kartę komuś innemu oraz wyświetlanie przez metodę **to_s**.

```
   def clean
       @cards = []
   end

   def give(card, other_hand)
       @cards.remove(card)
       other_hand.add(card)
   end
```

Metoda **to_s** zwraca string, który reprezentuje wszystkie karty gracza. Jeżeli atrybut cards nie zawiera elementów, zwracamy informację, że kolekcja jest pusta. W przeciwnym razie wykorzystujemy **map**, iterujemy po **obiektach klasy Card** zawartych w obiekcie **klasy Hand** w wyniku czego otrzymujemy tablicę,

a tę możemy już łączyć w łańcuch metodą **join**.

```
    def to_s
        return "empty" if @cards.empty?
        @cards.map { |card| card.to_s }.join(" ")
    end
```

Cały kod klasy ręka wygląda następująco:

```
class Hand
    def initialize
        @cards = []
    end

    def add(card)
        @cards.append(card)
    end

    def clean
        @cards = []
    end

    def give(card, other_hand)
        @cards.delete(card)
        other_hand.add(card)
    end

    def to_s
        return "empty" if @cards.empty?
        @cards.map { |card| card.to_s }.join(" ")
    end
end
```

## Grupowanie obiektów

Możemy stworzyć kilka obiektów klasy Card i przy użyciu obiektu klasy Hand je zgrupować.

```
hand = Hand.new
puts hand.to_s

card1 = Card.new("Q", "♣")
card2 = Card.new("A", "♦")
card3 = Card.new("6", "♥")
card4 = Card.new("3", "♠")
hand.add(card1)
hand.add(card2)
hand.add(card3)
hand.add(card4)

puts hand.to_s
```

Nasza kolekcja to: Q♣ 2♦ 6♥ 3♠

Przetestujmy przekazywanie kart. Tworzymy drugi obiekt klasy Hand. Przez metodę **give** obiektu hand, przekazujemy dwie karty z obiektu hand do nowego obiektu.

```
hand2 = Hand.new
hand.give(card2, hand2)
hand.give(card3, hand2)

puts "Hand 1: " + hand.to_s
puts "Hand 2: " + hand2.to_s
```

Warto jeszcze przetestować metodę clean:

```
hand.clean
puts hand.to_s
```

## Talia

Talia zawiera zbiór 52 kart. Jeżeli przyjrzymy się uważnie, talia posiada wiele cech wspólnych z ręką gracza. Tak jak ręka, zawiera kolekcję kart. Można do niej dodawać karty, a także przekazywać karty innym graczom.

Możemy to wykorzystać i zastosować dziedziczenie:

```
class Deck < Hand
end

deck = Deck.new
puts deck.to_s
```

Klasa potomna Deck **dziedziczy** atrybuty i metody z **klasy nadrzędnej** Hand, tzn. posiada wszystkie metody klasy nadrzędnej.

Talia posiada wszystkie możliwe kombinacje kart.

Co więcej, będziemy potrzebować metodę, która przetasuje karty oraz metodę, która rozdaje karty, dlatego na razie w klasie zostawimy sobie puste definicje tych metod.

```
  def shuffle
      #TODO
  end

  def deal
      #TODO
  end
```

Jak wiemy talia to gracz, który na start zna wszystkie swoje karty. Skoro tak, musimy talię wypełnić kartami. Dla wszystkich kolorów z klasy Card chcemy stworzyć wszystkie możliwe figury i dodać je do zbioru kart .Możemy zrobić to pętlą w pętli:

```
  def fill
      Card::SUITS.each do |s|
          Card::RANKS.each do |r|
              self.add(Card.new(r, s))
          end
      end
  end
```

*Ciekawostka: Istnieje w ruby metoda **`product`**, która zwraca iloczyn kartezjański dwóch tablic.*

Metoda **`fill`** wypełnia kolekcję kartami. Generujemy w pętli wszystkie możliwe kombinacje wartości z list **`Card.SUITS`** i **`Card.RANKS`**. Dla każdej kombinacji metoda tworzy nowy obiekt klasy Card, który dodaje do tablicy za pomocą metody **`add`** z klasy nadrzędnej.

Przetestujmy ją:

```
deck = Deck.new
deck.fill
puts deck.to_s
```

Wszystkie karty są wyświetlane wg. kolejności generowania.

Przejdźmy do definicji metody **`shuffle`**, która przetasuje karty w talii.

```
  def shuffle
      @cards.shuffle!
  end
```

Nie mamy tutaj trudnego zadania, gdyż możemy skorzystać z metody dostępnej w języku Ruby - **`shuffle`**, która przemieszcza elementy listy, ustawiając je w przypadkowej kolejności. Dodanie wykrzyknika na końcu sprawia, że metoda wykonuje się w miejscu i aktualizuje zawartość tablicy kartami **`@cards.`**

```
deck.shuffle
puts deck.to_s
```

Wszystkie elementy atrybutu **@cards** zostały potasowane.

Została metoda, która rozdaje karty do rąk graczy. Metoda powinna na starcie przyjąć listę obiektów, którym należy rozdać karty, oraz po ile kart powinien otrzymać każdy z nich.

```
  def deal(hands, to_hand=1)
      (0...to_hand).each do |_|
          hands.each do |hand|
              if @cards.size > 0
                  first_card = @cards[0]
                  self.give(first_card, hand)
              else
                  puts "No more cards in deck"
                  return
              end
          end
      end
  end
```

Metoda rozdaje każdemu graczowi po jednej karcie z talii.

Za każdym razem brana jest pierwsza karta z kolekcji kart. Karty rozdawane są dopóki w tali nie skończą się karty. Czynność tę będziemy wykonywać w zależności od tego ile kart ma trafić do każdego gracza przy rozdaniu (**to_hand**, domyślnie wartość 1).

Możemy przetestować, czy metoda działa prawidłowo. Musimy utworzyć i wypełnić talię kartami. Utworzyć 2 ręce graczy:

```
deck = Deck.new
deck.fill
deck.shuffle

hand1 = Hand.new
hand2 = Hand.new
hands = [hand1, hand2]
```

Każdy gracz otrzyma do ręki po 3 karty, co możemy sprawdzić wyświetlając wyniki.

```
deck.deal(hands, to_hand=3)
puts hand1
puts hand2
puts hand2
```

## Zakryta karta

Krupier na początku gry zakrywa swoją kartę. Chwilowo zaasymulujmy podobną sytuację.

Klasa **ReversibleCard** będzie oprócz koloru i rangi posiadała dodatkowy atrybut - czy karta odwrócona? W takim razie nie chcemy całkowicie nadpisać konstruktora z klasy Card. Za pomocą metody **super** najpierw wywołamy konstruktor z klasy nadrzędnej, a dopiero później stan karty.

```
class ReversibleCard < Card
   def initialize(rank, suit, known)
       super(rank, suit)
       @known = known
   end
end

card = ReversibleCard.new("A", "♥", false)
puts card.to_s
```

W zależności od stanu musimy też zmienić wyświetlanie karty:

```
  def to_s
       return "XX" unless @known
       super
   end
```

Dodamy jeszcze metodę, która pozwala odwrócić kartę:

```
  def reverse
      @known = !@known
  end
```

Przetestujmy ją:

```
card = ReversibleCard.new("A", "♥", false)
puts card.to_s
card.reverse
puts card.to_s
```

Wszystkie dwie klasy kart wyglądają o tak:

```
class Card
  SUITS = ["♥", "♦", "♣", "♠"]
  RANKS = ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q",
"K"]
   def initialize(rank, suit)
      @rank = rank
      @suit = suit
  end
   def to_s
      @rank + @suit
  end
end

class ReversibleCard < Card
  def initialize(rank, suit, known)
      super(rank, suit)
      @known = known
  end

  def to_s
      return "XX" unless @known
      super
  end
```

```
   def reverse
       @known = !@known
   end
end
```

## BlackJack

Ten długi wstęp doprowadził nas do momentu, gdy mamy wszystkie elementy, które odwzorowują nam przedmioty niezbędne do zagrania w BlackJacka.

Utworzę sobie plik **base_elements.rb**, w którym umieszczę klasy Card, Hand, Deck.

Klasa Card będzie zawierała funkcjonalności ReversibleCard z poprzedniej sekcji, dzięki czemu ReversibleCard możemy usunąć.

Wszystkie te elementy są uniwersalne nadają się do wykorzystania w wielu grach karcianych np. makao czy gry w wojnę (np. jako dodatkowych projektach).

**base_elements.rb**

```
class Card
   SUITS = ["♥", "♦", "♣", "♠"]
   RANKS = ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q",
"K"]
    def initialize(rank, suit, known=true)
       @rank = rank
       @suit = suit
       @known = known
   end

   def to_s
```

```
        return "XX" unless @known
        @rank + @suit
    end

    def reverse
        @known = !@known
    end
end

class Hand
    def initialize
        @cards = []
    end

    def add(card)
        @cards.append(card)
    end

    def clean
        @cards = []
    end

    def give(card, other_hand)
        @cards.delete(card)
        other_hand.add(card)
    end

    def to_s
        return "empty" if @cards.empty?
        @cards.map { |card| card.to_s }.join(" ")
    end
end

class Deck < Hand
```

```
    def fill
        Card::SUITS.each do |s|
            Card::RANKS.each do |r|
                self.add(Card.new(r, s))
            end
        end
    end

    def shuffle
        @cards.shuffle!
    end

    def deal(hands, to_hand=1)
        (0...to_hand).each do |_|
            hands.each do |hand|
                if @cards.size > 0
                    first_card = @cards[0]
                    self.give(first_card, hand)
                else
                    puts "No more cards in the deck"
                    return
                end
            end
        end
    end
end
```

Plik możemy zaimportować do pliku **main.rb** za pomocą **require** (wtedy musimy podać ścieżkę) lub **require_relative** (podajemy nazwę pliku), gdy plik znajduje się w tym samym folderze.

```
require_relative "base_elements"
```

Sprawdź czy zaimportowanie działa tworząc dowolne obiekty korzystające z klas bazowych.

```
card = Card.new("Q","♣")
puts card.to_s
```

## Elementy gry w BlackJacka

Karty w BlackJacku mają z góry narzucone wartości punktowe, czego nie mają domyślnie zaimplementowanego. Możemy stworzyć klasę dziedziczącą z klasy kart, która będzie pozwalała odczytać wartość danej karty.

```
class BlackJackCard < Card
   ACE = 1

   def value
   end
end
```

As to karta, która przyjmuje wartość 1 albo 11, ta informacja przyda nam się później.

Każdorazowo musimy sprawdzić czy karta jest nam znana - tzn jej wartość jest położona widoczną stroną.

```
   def value
       return nil unless @known

       points = RANKS.index(@rank) + 1
       points = 10 if points > 10
       points
   end
```

Punkty kartom nadajemy na podstawie ich indeksu w tablicy **RANKS**.

Pierwszą kartą na naszej tablicy RANKS jest as, będzie miał indeks 0, po dodaniu 1, za asa otrzymamy punkt, za np. 2 kier 2 punkty i tak,  pokolei dla każdej karty, aż do figur (J, Q, K). Karty powyżej 10 indeksu, czyli figury, dodają graczowi zawsze 10 punktów.

Możemy przetestować tworząc obiekt:

```
card = BlackJackCard.new("Q", "♥")
puts card.value
```

W grze będziemy korzystać z klasy Hand, z pliku base_elements.rb rozszerzonej o dodatkową informację - imię gracza. Skorzystamy z konstruktora klasy nadrzędnej za pomocą **super**.

```
class BlackJackHand < Hand
   def initialize(name)
       super()
       @name = name
   end
end
```

Zwróć uwagę, że **konstruktor klasy Hand** nie przyjmuje żadnych argumentów, dlatego wywołując metodę **super()** spełniamy ten warunek (porównaj z wywołaniem **super** bez nawiasów).

Potrzebujemy też informację o punktach za karty posiadane w ręce. Po pierwsze wiemy, że jeśli w ręce mamy karty zakryte, to nie możemy obliczyć ich wartości i zwrócimy wartość nil.

```
   def total
       return nil if @cards.any? { |card| card.value == nil }
   end
```

Następnie sumujemy wartości kart.

```
   def total
       return nil if @cards.any? { |card| card.value == nil }

       @cards.map(&:value).sum
   end
```

Przetestujmy metodę

```
hand = BlackJackHand.new('John')
card1 = BlackJackCard.new("Q", "♥")
card2 = BlackJackCard.new("A", "♥")
card3 = BlackJackCard.new("4", "♥")
hand.add(card1)
hand.add(card2)
hand.add(card3)
puts hand.to_s
puts "Points in hand: #{hand.total}"
```

Wygląda, że liczy punkty prawidłowo, jednak wystarczyłoby, dobrać jeszcze jedną kartę o wartości 6 w górę, by przekroczyć 21 i przegrać grę. Dlatego musimy sprawdzić czy w zbiorze znajduje się as. Następnie dodać punkty za asa jeśli nam się to opłaca, tzn jeżeli brakuje nam mniej lub dokładnie 10 punktów do 21.

```
  def total
      return nil if @cards.any? { |card| card.value == nil }

      points = @cards.map(&:value).sum

      ace_in_hand = true if @cards.any? do |card|
          card.value == BlackJackCard::ACE
      end

      points += 10 if ace_in_hand && points <= 10
      points
  end
```

Dodajemy tylko 10 a nie 11, ponieważ już dodaliśmy 1 punkt za asa.

Następnie chcemy zmienić metodę **to_s**, aby wyświetlać sumaryczną wartość punktową ręki:

```
  def to_s
      hand = @name + " - " + super
      hand += " (points: #{total})"
```

```
        hand
    end
```

Trzeba jeszcze sprawdzić, czy po dobraniu kolejnej kart w ręce mamy furę.

```
    def is_busted?
        total > 21
    end
```

Cała klasa wygląda następująco:

```
class BlackJackHand < Hand
    def initialize(name)
        super()
        @name = name
    end

    def to_s
        hand = @name + " - " + super
        hand += " (points: #{total})"
        hand
    end

    def total
        return nil if @cards.any? { |card| card.value == nil }

        points = @cards.map(&:value).sum

        ace_in_hand = true if @cards.any? do |card|
            card.value == BlackJackCard::ACE
        end

        points += 10 if ace_in_hand && points < 11
        points
    end
```

```
    def is_busted?
        total > 21
    end
end
```

Przyda nam się jeszcze klasa gracza, która dziedziczy metody klasy **`BlackJackHand`** a oprócz tego pokaże informacje jak kończy się dla danego gracza rozgrywka - czy wygrywa, przegrywa, czy może jest remis. Dodatkowo potrzebujemy metody, która spowoduje przegraną gracza, który otrzyma furę (*ang. bust*).:

```
class Player < BlackJackHand
    def bust
        puts "Player #{@name} busts"
        lose
    end

    def lose
        puts "Player #{@name} loses"
    end

    def win
        puts "Player #{@name} wins"
    end

    def draw
        puts "Player #{@name} has a draw"
    end
end
```

Przetestujmy gracza:

```
player = Player.new('John')
puts player
player.bust
```

Warto dodać jeszcze jedną metodę gracza - pytanie czy chce dobrać kartę

```
  def take_a_card?
      print "#{@name} - do you want to take a card? (y/n): "
      answer = gets.chomp
      return answer == "y"
  end
```

Cała klasa Player wygląda tak:

```
class Player < BlackJackHand
  def bust
      puts "Player #{@name} busts"
      lose
  end

  def lose
      puts "Player #{@name} loses"
  end

  def win
      puts "Player #{@name} wins"
  end

  def draw
      puts "Player #{@name} has a draw"
  end

  def take_a_card?
      print "#{@name} - do you want to take a card? (y/n): "
      answer = gets.chomp
      return answer == "y"
  end
end
```

Mamy już zdefiniowanych graczy, teraz potrzebny jest rozdający. Podobnie do gracza ma pewne karty w ręku, więc wykorzystamy dziedziczenie:

```
class Dealer < BlackJackHand
end
```

Rozdający bierze dodatkowe karty dopóki ma mniej niż 17 punktów.

```
class Dealer < BlackJackHand
   def take_a_card?
       total < 17
   end

   def bust
       puts "Dealer busts"
   end

   def reverse_first_card
       first_card = @cards[0]
       first_card.reverse
   end
end
```

Metoda **bust** wyświetla informację, gdy rozdający ma furę. Za pomocą metody **reverse_first_card** odwracamy pierwszą kartę rozdającego.

Zostaje nam stworzenie dedykowanej talii, która będzie tworzyła karty do BlackJacka

```
class BlackJackDeck < Deck
   def fill
       Card::SUITS.each do |s|
           Card::RANKS.each do |r|
               self.add(BlackJackCard.new(r, s))
           end
       end
   end
end
```

## Tworzenie obiektu gry

Żeby powiązać wszystkie klasy ze sobą potrzebujemy klasy do gry. Nazwiemy ją **BlackJackGame**:

```
class BlackJackGame
end
```

Na start potrzebujemy listę imion graczy i dla każdego gracza tworzymy jego obiekt w grze. W konstruktorze tworzymy też rozdającego oraz talię.

```
class BlackJackGame
   def initialize(names)
       @players = []
       names.each do |name|
           player = Player.new(name)
           @players.append(player)
       end

       @dealer = Dealer.new("Dealer")
       @deck = BlackJackDeck.new
       @deck.fill
       @deck.shuffle
   end
end
```

W grze będziemy sprawdzać, którzy gracze jeszcze grają, nie zebrali fury

```
   def not_busted_players
       @players.each_with_object([]) do |player, arr|
           arr.append(player) unless player.is_busted?
       end
   end
```

Stworzymy metodę, która rozdaje dodatkowe karty wszystkim graczom i rozdającemu. Metodę wywołamy na obiekcie klasy lub Player lub Dealer

Rozdawać karty możemy dopóki gracz lub rozdający chcą otrzymywać karty oraz w ręce ich nie uzbiera się suma fury.

```
def additional_cards(user)
    while user.take_a_card? && (not user.is_busted?) do
      @deck.deal([user])
      puts user.to_s
      user.bust if user.is_busted?
    end
end
```

W przypadku tych dwóch wywołań metod zadziała polimorfizm. Wywołanie metody **user.is_take_a_card?** zadziała niezależnie od tego, czy parametr **user** jest obiektem klasy Player czy klasy Dealer. Obie implementują metodę o tej samej nazwie. To samo dotyczy metody odziedziczonej z klasy BlackJackHand - **user.is_busted?**

## Metoda play

Napiszmy sobie najpierw plan naszej rozgrywki na podstawie opisu gry.

Na początek każdy gracz i rozdający dostają po 2 karty
Rozdający zasłania kartę
Dla każdego gracza
  Dopóki gracz dobiera kartę i nie ma fury
    Przydziel graczowi kartę
Jeśli nie ma już graczy pozostając w grze
  Odsłoń karty rozdającego
W przeciwnym razie
  Dopóki rozdający dobiera karty i nie ma fury
    Przydziel rozdającemu kartę
      Jeśli rozdający ma furę
        Dla każdego gracza w grze
          Gracz wygrywa
        W przeciwnym razie
          Dla każdego gracza pozostającego w grze
            Gdy suma punktów gracza jest większa niż rozdającego
              Gracz wygrywa

Gdy suma gracza jest mniejsza rozdającego
  Gracz przegrywa
W przeciwnym razie
  Gracz remisuje

Możemy ten opis potraktować bardzo dosłownie i tak też zapisać naszą rozgrywkę:

```
def play
    users = @players + [@dealer]
    @deck.deal(users, per_hand = 2)
    @dealer.reverse_first_card
    @players.each { |player| puts player.to_s }
    puts @dealer.to_s
    @players.each { |player| give_additional_cards(player) }
    @dealer.reverse_first_card

    if not_busted_players.empty?
        puts @dealer.to_s
        puts "Dealer wins"
    else
        puts @dealer.to_s
        give_additional_cards(@dealer)
        if @dealer.is_busted?
            not_busted_players.each { |player| player.win }
        else
            not_busted_players.each do |player|
                if player.total > @dealer.total
                    player.win
                elsif player.total < @dealer.total
                    player.lose
                else
                    player.draw
                end
            end
        end
```

```
        end
        @players.each { |player| player.clean }
        @dealer.clean
    end
```

Możemy przetestować naszą grając dwoma wirtualnymi graczami

```
game = BlackJackGame.new(['John','Anna'])
game.play
```

## Rozgrywka

Sprawmy, by nasza gra rzeczywiście przypominała grę konsolową, w której można podać imiona graczy.

Poza naszymi klasami stworzymy metodę, która rozpoczyna rozgrywkę - pyta o liczbę graczy oraz ich imiona, a następnie uruchamia klasę z grą.

```
def run_game
    print("------ BlackJack Game ------\n")

    puts "Number of players (1 - 7): "
    number = gets.chomp.to_i

    names = []
    (1..number).each do |nr|
        puts "Player #{nr} name:"
        name = gets.chomp
        names.append(name)
    end
    game = BlackJackGame.new(names)
    game.play()
end
```

Uruchamiamy ją przez wpisanie nazwy. Super nasza gra działa!

```
run_game
```

Co jeśli chcielibyśmy zagrać ponownie?

Musimy wywoływanie metody **play()** zapętlić. Możemy dodać pętlę while, którą przerwiemy jeśli użytkownik nie będzie chciał kontynuować.

```
  while true do
       game.play()
       puts "Do you wanna play again (y/n)"
       again = gets.chomp
       break unless again == "y"
  end
```

W tym momencie nasza metoda **run_game** wygląda tak:

```
def run_game
   print("------ BlackJack Game ------\n")

   puts "Number of players (1 - 7): "
   number = gets.chomp.to_i

   names = []
   (1..number).each do |nr|
       puts "Player #{nr} name:"
       name = gets.chomp
       names.append(name)
   end
    game = BlackJackGame.new(names)
   while true do
       game.play()
       puts "Do you wanna play again (y/n)"
       again = gets.chomp
       break unless again == "y"
   end
```

```
   puts "----- Have a nice day! -----"
end

run_game
```

## Finalny projekt

W naszym pliku **main.rb** mamy wszystkie klasy związane z grą oraz na końcu metodę **run_game**.

Możemy nasz logicznie podzielić kod na pliki

- base_elements.rb,
- main.rb
- blackjack.rb

Plik main.rb będzie zawierał tylko metodę potrzebną do rozpoczęcia gry i importowanie zawartości pliku blackjack. W pliku blackjack niech znajdują się wszystkie klasy charakterystyczne dla gry w BlackJacka. Plik base_elements.rb pozostaje bez zmian.

`main.rb`

```
require_relative "blackjack"

def run_game
   print("------ BlackJack Game ------\n")

   puts "Number of players (1 - 7): "
   number = gets.chomp.to_i

   names = []
   (1..number).each do |nr|
       puts "Player #{nr} name:"
       name = gets.chomp
       names.append(name)
```

```
  end
   game = BlackJackGame.new(names)
  while true do
      game.play()
      puts "Do you wanna play again (y/n)"
      again = gets.chomp
      break unless again == "y"
  end

  puts "----- Have a nice day! -----"
end

run_game
```

blackjack.rb

```
 require_relative "base_elements"

class BlackJackCard < Card
  attr_reader :value
  ACE = 1

  def value

      return nil unless @known

      points = RANKS.index(@rank) + 1
      points = 10 if points > 10
      points
  end
end

class BlackJackHand < Hand
```

```
    def initialize(name)
        super()
        @name = name
    end

    def to_s
        hand = @name + " - " + super
        hand += " (points: #{total})"
        hand
    end

    def total
        return nil if @cards.any? { |card| card.value == nil }

        points = @cards.map(&:value).sum

        ace_in_hand = true if @cards.any? do |card|
            card.value == BlackJackCard::ACE
        end

        points += 10 if ace_in_hand && points < 11
        points
    end

    def is_busted?
        total > 21
    end
end

class BlackJackDeck < Deck
    def fill
        Card::SUITS.each do |s|
            Card::RANKS.each do |r|
                self.add(BlackJackCard.new(r, s))
```

```
            end
        end
    end
end

class Player < BlackJackHand
    def bust
        puts "Player #{@name} busts"
        lose
    end

    def lose
        puts "Player #{@name} loses"
    end

    def win
        puts "Player #{@name} wins"
    end

    def draw
        puts "Player #{@name} has a draw"
    end

    def take_a_card?
        print "#{@name} - do you want to take a card? (y/n): "
        answer = gets.chomp
        return answer == "y"
    end
end

class Dealer < BlackJackHand
    def take_a_card?
        total < 17
    end
```

```
  def bust
      puts "Dealer busts"
  end

  def reverse_first_card
      first_card = @cards[0]
      first_card.reverse
  end
end

class BlackJackGame
  def initialize(names)
      @players = []
      names.each do |name|
          player = Player.new(name)
          @players.append(player)
      end

      @dealer = Dealer.new("Dealer")
      @deck = BlackJackDeck.new
      @deck.fill
      @deck.shuffle
  end

  def not_busted_players
      @players.each_with_object([]) do |player, arr|
          arr.append(player) unless player.is_busted?
      end
  end

  def give_additional_cards(user)
      while user.take_a_card? && (not user.is_busted?) do
        @deck.deal([user])
```

```
      puts user.to_s
      if user.is_busted?
        user.bust
        break
      end
    end
end

def play
    users = @players + [@dealer]
    @deck.deal(users, per_hand = 2)
    @dealer.reverse_first_card
    @players.each { |player| puts player.to_s }
    puts @dealer.to_s
    @players.each { |player| give_additional_cards(player) }
    @dealer.reverse_first_card

    if not_busted_players.empty?
        puts @dealer.to_s
        puts "Dealer wins"
    else
        puts @dealer.to_s
        give_additional_cards(@dealer)
        if @dealer.is_busted?
            not_busted_players.each { |player| player.win }
        else
            not_busted_players.each do |player|
                if player.total > @dealer.total
                    player.win
                elsif player.total < @dealer.total
                    player.lose
                else
                    player.draw
                end
```

```
                end
            end
        end
        @players.each { |player| player.clean }
        @dealer.clean
    end
end
```

Oczywiście to nie koniec możliwości rozbudowywania projektu można dołożyć do niego obsługę błędów, czy własne funkcjonalności jak np. obstawianie żetonów.

Zachęcam, by ten projekt był tylko pierwszym z wielu, które napiszesz w języku Ruby.