

Kółko i krzyżyk

Stwórz grę w kółko i krzyżyk dla 2 graczy. Przykładowa rozgrywka mogłaby wyglądać ten sposób:

```
  A  B  C
0  . | . | .
1  . | . | .
2  . | . | .
```

Player X, mark position: 1A

```
  A  B  C
0  X | . | .
1  . | . | .
2  . | . | .
```

Player O, mark position: 2B

```
  A  B  C
0  X | . | .
1  . | O | .
2  . | . | .
```

Player X, mark position: 1B

...

```
  A  B  C
0  X | O | X
1  X | O | X
2  O | X | O
```

Draw!

Spróbuj samodzielnie rozwiązać to zadanie, a następnie dodaj własne pomysły, rozszerzenia np. imiona graczy, obsługę błędów.

Rozwiązanie

Zacniemy od rozpisania kroków jakie zazwyczaj gracze robią, grając w tradycyjną wersję gry:

1. Narysowanie planszy
2. Gracz pierwszy oznacza swój ruch znakiem X
3. Gracz drugi oznacza swój ruch znakiem O
4. Gra trwa, dopóki któryś z graczy nie uzyska trzech znaków X lub O tworzących linię (horyzontalnie, wertykalnie lub na skos).

Mamy, więc takie czynności jak:

- Wyświetlenie stanu planszy
- Ruchy graczy
- Sprawdzenie wyniku

Utwórz plik **tictactoe.rb**.

Stworzenie planszy

Plansza na kartce to przecinające się linie. W naszym przypadku będzie to prosta tablica 2-wymiarowa wypełniona kropkami oznaczającymi wolne pola.

Nasz kod wyglądałby na tę chwilę następująco

```
board = [  
  ['.', '.', '.'],  
  ['.', '.', '.'],  
  ['.', '.', '.']  
]  
p board
```

Metoda **p** wyświetla zawartość tablicy.

Widzimy, że wynik nie przypomina planszy do gry w kółko i krzyżyk, dlatego musimy zmienić sposób wyświetlania. Wykorzystamy do tego pętlę.

```
board.each { |row| p row }
```

Nadal jednak nie mamy informacji o numerze wyświetlanego wiersza, dlatego skorzystamy z pętli **each_with_index**:

```
board.each_with_index do |row, index|
  p index, row
end
```

Jeszcze poprawimy wyświetlanie za pomocą :

```
board.each_with_index do |row, index|
  p index.to_s + " " + row.join(" ")
end
```

Oraz nazwiemy kolumny:

```
puts "  A  B  C"
board.each_with_index do |row, index|
  puts index.to_s + " " + row.join(" | ")
end
```

W efekcie w terminalu pojawia się plansza gry:

```
  A  B  C
0 . | . | .
1 . | . | .
2 . | . | .
```

Wykonanie ruchu

Do elementów tablicy możemy odwoływać się po indeksach. Żeby edytować element podajemy jego wartości **[wiersz][kolumna]**.

```
board[1][0] = 'X'

puts "  A  B  C"
board.each_with_index do |row, index|
  puts index.to_s + " " + row.join(" | ")
end
```

Będziemy w czasie gry wielokrotnie wyświetlać planszę, dlatego warto tę część

kodu przenieść do metody **show**, która jako parametr przyjmie naszą tablicę z planszą.

```
def show(board)
  puts "  A  B  C"
  board.each_with_index do |row, index|
    puts index.to_s + " " + row.join(" | ")
  end
end

board = [
  ['. ', '. ', '. '],
  ['. ', '. ', '. '],
  ['. ', '. ', '. ']
]

show(board)
board[1][0] = 'X'
show(board)
```

Czas stworzyć graczy i pozwolić podać im swoją pozycję w formacie wiersz kolumna(np. 1A, 3B, 0C itp).

W miejsce `board[1][0] = 'X'` wprowadzimy dane od użytkownika.

```
print "Player X, mark position: "
position = gets

puts position
```

Zwróć uwagę, że w tym miejscu celowo używamy metody `print`, ponieważ zależy nam, by użytkownik wprowadził swój wybór w tej samej linii, w której pojawia się komunikat.

Format wprowadzania pozwala nam wyciągnąć i łatwo rozdzielić wiersz od kolumny za pomocą metody **split**.

```
row, col = position.split("")
```

O ile informacja w zmiennej **row** zawiera tekst z numerem wiersza, to w przypadku kolumny musimy znaleźć sposób na przypisanie wartości numerycznej naszym literom.

Możemy na przykład utworzyć sobie tablicę pomocniczą z kolumnami, a następnie na jej podstawie oznaczyć odpowiedni numer indeksu:

```
columns = ["A", "B", "C"]
row, col = position.split("")
p row.to_i, columns.index(col)
```

Za pomocą indeksów bez problemu możemy już oznaczyć odpowiedni znak na naszej planszy.

Dodajemy **col.upcase**, dzięki czemu nawet jeśli użytkownik poda małą literę, to dopasujemy ją do odpowiedniej kolumny.

```
print "Player X, mark position: "
position = gets.chomp
columns = ["A", "B", "C"]
row, col = position.split("")
row, col = row.to_i, columns.index(col.upcase)
board[row][col] = "X"
show(board)
```

Ten fragment kodu również będzie wykorzystywać wielokrotnie, dlatego utworzymy metodę ruchu gracza:

```
def move(player, board)
  print "Player #{player}, mark position: "
  position = gets.chomp
  columns = ["A", "B", "C"]
  row, col = position.split("")
  row, col = row.to_i, columns.index(col.upcase)
  board[row][col] = player

  return board # return na końcu metody opcjonalnie
end
```

Na chwilę obecną nasz cały kod wygląda następująco:

```
def show(board)
  puts "  A  B  C"
  board.each_with_index do |row, index|
    puts index.to_s + " " + row.join(" | ")
  end
end

def move(player, board)
  print "Player #{player}, mark position: "
  position = gets.chomp
  columns = ["A", "B", "C"]
  row, col = position.split("")
  row, col = row.to_i, columns.index(col.upcase)
  board[row][col] = player

  return board
end

board = [
  ['.', '.', '.'],
  ['.', '.', '.'],
  ['.', '.', '.']
]

show(board)
board = move("X", board)
show(board)
board = move("O", board)
show(board)
```

Wykonywanie ruchów w pętli

Stworzymy na start prostą pętlę, w której gracze będą wykonywać kolejne ruchy.

Ile minimalnie ruchów musi zostać wykonane przez gracza, aby najszybciej można było zdecydować, że nastąpiła wygrana, któregoś z graczy?

Odpowiedź to 3.

Jeżeli w ciągu 3 ruchów gracz ustawi linie prostą ze swojego znaku, to gra może się skończyć.

```
move_counter = 0
while move_counter < 3 do
  board = move("X", board)
  show(board)
  board = move("O", board)
  show(board)
  move_counter += 1
end
```

Nie chodzi o 3 ruchy obu graczy. Tak naprawdę, wystarczy co najmniej 3 ruchy pierwszego gracza (i 2 ruchy drugiego).

Zamiast za każdym razem wymuszać wykonanie obu ruchów możemy w pętli naprzemiennie pytać graczy o ruch.

Zaczyna gracz X, dlatego utworzymy zmienną **x** o wartości początkowej **true**. Jeśli w danej rundzie **x** ma wartość **true** wiemy, że jest to ruch należący do pierwszego gracza, w przeciwnym wypadku, znak stawia gracz O. Po każdym ruchu wyświetlamy tablicę i odwracamy wartość zmiennej **x**

```
move_counter = 0
x = true
while move_counter < 5 do
  if x == true
    move("X", board)
```

```

else
    move("O", board)
end
show(board)
move_counter += 1
x = !x
end

```

Zwróć uwagę, że teraz **move_counter** liczy ruchy każdego gracza, dlatego min. liczba pętli to 5.

Sprawdzenie stanu aktualnego gracza możemy przenieść do osobnej metody, przyjmującej argument z wartością **true/false**. Jeśli zmienna **player_x** będzie miała wartość **true** chcemy zwrócić znak "X", w przeciwnym razie znak "O".

```

def current_player(player_x)
    return "X" if player_x
    "O"
end

p current_player(false)

```

Możemy tę metodę uprościć do postaci:

```

def current_player(player_x)
    player_x ? "X" : "O"
end

```

W efekcie w naszej pętli zaszły zmiany:

```

x = true
move_counter = 0
while move_counter < 5 do
    move(current_player(x), board)
    show(board)
    move_counter += 1
    x = !x
end

```

Nasz cały obecny kod wygląda następująco:

```
def show(board)
  puts "  A   B   C"
  board.each_with_index do |row, index|
    puts index.to_s + " " + row.join(" | ")
  end
end

def move(player, board)
  print "Player #{player}, mark position: "
  position = gets.chomp
  columns = ["A", "B", "C"]
  row, col = position.split("")
  row, col = row.to_i, columns.index(col.upcase)
  board[row][col] = player
  board
end

def current_player(player_x)
  player_x ? "X" : "O"
end

board = [
  ['.', '.', '.'],
  ['.', '.', '.'],
  ['.', '.', '.']
]

show(board)

x = true
move_counter = 0
while move_counter < 5 do
```

```

    move(current_player(x), board)
    show(board)
    move_counter += 1
    x = !x
end
show(board)

```

Warunek wygranej

Trzy ruchy pierwszego z graczy to minimalna liczba, jaka musi się odbyć, by gracz miał szansę wygrać. Jednak może zdarzyć się tak, że żaden z użytkowników w tym czasie nie ułoży linii.

Są 3 sposoby na wygraną:

- ułożenie linii w poziomie - wiersza
- ułożenie linii w pionie - kolumny
- ułożenie linii po przekątnej

Ułożenie linii w poziomie

Dla uproszczenia, odłożymy na chwilę nasz dotychczasowy kod i założmy, że w tej chwili po 5 ruchach mamy taką sytuację:

```

board = [
  ['.', 'O', '.'],
  ['X', 'X', 'X'],
  ['.', 'O', '.']
]

```

Chcemy więc sprawdzić czy dla każdego wiersza, wszystkie elementy są takie same i jeśli tak, to dowiedzieć się jaki znak mamy w tym wierszu:

```

board.each do |row|
  if row.uniq.count == 1
    p row[0]
  end
end

```

```
end
```

Wyświetlam pierwszy element - znak zwycięskiego gracza, gdyż jeśli warunek jest spełniony, to i tak wszystkie elementy będą takie same.

Co jeśli jednak tablica będzie wyglądała o tak:

```
board = [  
  ['X', 'O', 'X'],  
  ['.', '.', '.'],  
  ['X', '.', 'O']  
]
```

Musimy sprawdzić, czy rzeczywiście znaleziony znak to X lub O:

```
board.each do |row|  
  if row.uniq.count == 1 && row[0] != "."  
    p row[0]  
  end  
end
```

Super, mamy to! Teraz nasza pętla wyświetli element tylko jeśli rzeczywiście znajdzie zwycięzcę.

Ułożenie linii w pionie

Rozpatrzmy przypadek drugi, kiedy zwycięzca utworzył linię w pionie:

```
board = [  
  ['.', 'X', '.'],  
  ['O', 'X', 'O'],  
  ['.', 'X', '.']  
]
```

Wiersze w planszy są gotowymi tablicami i widać to było na pierwszy rzut oka, ale skąd wziąć kolumny?

Pewnie powiecie, że przecież możemy wyciągnąć każdą kolumnę osobno.. Wystarczy w pętli przejść kolejne wiersze i wyciągnąć odpowiedni element na

podstawie indeksu i dodać go do nowej tablicy kolumn. Mając tablicę, reszta kodu zostaje jak w przypadku sprawdzenia w poziomie - sprawdzamy czy elementy w tablicy są takie same:

```
(0..2).each do |col|
  column = []
  board.each do |row|
    column.push(row[col])
  end
  if column.uniq.count == 1 && column[0] != "."
    p column[0]
  end
end
```

Jest to poprawne rozwiązanie, ale ruby daje nam metodę **transpose**, która zamienia miejscami wiersze z kolumnami, dzięki czemu nasza pętla może być znacznie prostsza:

```
board.transpose.each do |col|
  if col.uniq.count == 1 && col[0] != "."
    p col[0]
  end
end
```

Ułożenie linii po przekątnej:

Oczywiście można to sprawdzić podając konkretne indeksy naszej planszy.

Chcemy sytuacji, która wygląda o tak (pseudokod):

- board[0,0] == board[1,1] == board[2,2]
- board[0,2] == board[1,1] == board[2,0]

Nasza plansza wyglądałaby następująco:

```
board = [
  ['X', '.', '.'],
  ['O', 'X', 'O'],
  ['.', '.', 'X']
]
```

Musimy zebrać, a następnie porównać elementy, których oba indeksy są takie same:

```
diag = []
(0..2).each do |index|
  diag.push(board[index][index])
end
if diag.uniq.count == 1 && diag[0] != "."
  p diag[0]
end
```

Kod ten nie zadziała jednak w przypadku przeciwnej przekątnej:

```
board = [
  ['.', '.', 'X'],
  ['O', 'X', 'O'],
  ['X', '.', '.']
]
```

Nic nie szkodzi, w ruby mamy do dyspozycji metodę **reverse**, która odwraca tablicę. Niewielka zmiana sprawi, że kod sprawdzający przeciwną przekątną będzie bardzo podobny do poprzedniego:

```
diag = []
(0..2).each do |index|
  diag.push(board.reverse[index][index])
end
if diag.uniq.count == 1 && diag[0] != "."
  p diag[0]
end
```

Znalezienie zwycięzcy

Wróćmy do naszej gry. Żeby stwierdzić czy użytkownik wygrał, trzeba sprawdzić wszystkie trzy przypadki. Przenieśmy powyższe fragmenty kodu do osobnych metod. Jednak zamiast zwracać konkretny znak, nasze metody będą jedynie odpowiadać **true/false**.

```
def win_horizontal(board)
  board.each do |row|
    if row.uniq.count == 1 && row[0] != "."
      return true
    end
  end
  false
end
```

```
def win_vertical(board)
  board.transpose.each do |col|
    if col.uniq.count == 1 && col[0] != "."
      return true
    end
  end
  false
end
```

```
def win_diagonal(board)
  diag = []
  (0..2).each do |index|
    diag.push(board[index][index])
  end
  if diag.uniq.count == 1 && diag[0] != "."
    return true
  end
end
```

```

end
# odwrotna przekątna
diag = []
(0..2).each do |index|
  diag.push(board.reverse[index][index])
end
if diag.uniq.count == 1 && diag[0] != "."
  return true
end
false
end

```

Piszemy kod DRY (*ang. don't repeat yourself*), dlatego pozbywamy się powtarzającej części kodu:

```

def check_line(arr)
  arr.uniq.count == 1 && arr[0] != "."
end

```

Metoda **check_line** zwróci nam wartość **true/false**, jeśli udało się ułożyć linię. Następnie wspólną metodę możemy umieścić we wcześniej stworzonych metodach

```

def win_horizontal(board)
  board.each do |row|
    return true if check_line(row)
  end
  false
end

```

```

def win_vertical(board)
  board.transpose.each do |col|
    return true if check_line(col)
  end
  false
end

```

```
end
```

```
def win_diagonal(board)
  diag = []
  (0..2).each do |index|
    diag.push(board[index][index])
  end
  return true if check_line(diag)
  # odwrotna przekątna
  diag = []
  (0..2).each do |index|
    diag.push(board.reverse[index][index])
  end
  return true if check_line(diag.reverse)
  false
end
```

Metodę **win_diagonal** możemy jeszcze trochę uprościć tworząc metodę create diagonal.

```
def create_diag(arr)
  diag = []
  (0..2).each do |index|
    diag.push(arr[index][index])
  end
  diag
end

def win_diagonal(board)
  # użycie create diagonal
  diag = create_diag(board)
  return true if check_line(diag)
  ...
end
```

Dzięki czemu cała metoda **win_diagonal** została odchudzona do 3 linii:

```
def create_diag(arr)
  diag = []
  (0..2).each do |index|
    diag.push(arr[index][index])
  end
  diag
end

def win_diagonal(board)
  return true if check_line(create_diag(board))
  return true if check_line(create_diag(board.reverse))
  false
end
```

Zostało nam wszystko połączyć w całość. Chcemy, by pętla wykonywała się tak długo, dopóki któryś z graczy nie wygra. Tutaj przyda nam się metoda, która sprawdzi 3 różne warunki wygranej.

```
def is_winner(board)
  if win_horizontal(board) || win_vertical(board) || win_diagonal(board)
    return true
  end
  false
end
```

Tę metodę wykorzystamy w pętli. Chcemy pozwolić użytkownikom grać, dopóki, któryś z nich nie wygra, albo nie osiągną maksymalnej liczby pól.

Pętlę **while** zmieniliśmy na **until**, gdyż bardziej naturalnie czyta się kod - "wykonuj dopóki liczba ruchów nie równa się liczbie wszystkich pól czyli 9".

Słowo kluczowe **break** wymusza wyjście z pętli, jeżeli aktualny gracz wygrał.

```

until move_counter == 9 do
  move(current_player(x), board)
  show(board)
  break if is_winner(board)
  move_counter += 1
  x = !x
end

```

Wyświetlanie wyniku

Na zakończenie musimy sprawdzić kto wygrał:

```

if is_winner(board)
  puts "Player #{current_player(x)} wins"
else
  puts "Draw"
end

```

Cały kod naszej gry kółko i krzyżyk wygląda tak:

```

def show(board)
  puts "  A  B  C"
  board.each_with_index do |row, index|
    puts index.to_s + " " + row.join(" | ")
  end
end

def move(player, board)
  print "Player #{player}, mark position: "
  position = gets.chomp
  columns = ["A", "B", "C"]
  row, col = position.split("")
  row, col = row.to_i, columns.index(col.upcase)
  board[row][col] = player
end

```

```

    board
end

def current_player(player_x)
  player_x ? "X" : "O"
end

def check_line(arr)
  arr.uniq.count == 1 && arr[0] != "."
end

def win_horizontal(board)
  board.each do |row|
    return true if check_line(row)
  end
  false
end

def win_vertical(board)
  board.transpose.each do |col|
    return true if check_line(col)
  end
  false
end

def create_diag(arr)
  diag = []
  (0..2).each do |index|
    diag.push(arr[index][index])
  end
  diag
end

```

```

def win_diagonal(board)
  return true if check_line(create_diag(board))
  return true if check_line(create_diag(board.reverse))
  false
end

def is_winner(board)
  if win_horizontal(board) || win_vertical(board) || win_diagonal(board)
    return true
  end
  false
end

board = [
  ['.', '.', '.'],
  ['.', '.', '.'],
  ['.', '.', '.']
]

show(board)

x = true
move_counter = 0
until move_counter == 9 do
  move(current_player(x), board)
  show(board)
  break if is_winner(board)
  move_counter += 1
  x = !x
end

if is_winner(board)

```

```
    puts "Player #{current_player(x)} wins"  
else  
    puts "Remis"  
end
```