Principles of Java Language with Applications, PIC20a
O. Azencot, originally created by E. Ryu
Spring 2019

UCLA

Homework 4
Due 5pm, Friday, May 24, 2019

In this homework assignment, you'll implement the game Connect Four.
`https://en.wikipedia.org/wiki/Connect_Four`
The board size is 7 columns by 6 rows. Use red and black for the two players and have red always go first.

Download the starter code `CFPlayer.java`, `CFGame.java`, and `Test.java`. Your code must work with `Test.java`. Put everything in the package `hw4`. Submit `CFGame.java`, `RandomAI.java`, `YourNameAI.java`, `ConsoleCF.java`, and `GUICF.java`.

Remark. When writing a `class`, it's usually a bad idea to place the `main` function in the same `class` you're trying to debug. There are several reasons for this, but one reason is that doing so will circumvent encapsulation.

Problem 1:

Finish writing the `class CFGame`.

The method

```
public boolean play(int c)
```

plays the column `c`. If column `c` cannot be played because it is full or because it is an invalid column, `return false`. Otherwise, `return true`. The columns are counted with 1-based indexing, i.e., the columns range from 1 to 7.

The method

```
public boolean isGameOver()
```

`returns true` if the game is over because there is a winner or because there are no more possible moves and `returns false` otherwise.

The method

```
public int winner()
```

`returns 1` if red is the winner, `-1` if black is the winner, and `0` if the game is a draw. This method should be called when `isGameOver returns true`.

Remark. `CFGame` provides the bare-bone game logic but doesn't play the game itself. By having `ConsoleCF` and `GUICF` inherit `CFGame` you avoid duplicating the code that implements the basic Connect Four game logic, while allowing them to play the Connect Four game in a different manner.

Problem 2:

Write the `class RandomAI` that implements `CFPlayer`.

The implementation of the method

```
int nextMove(CFGame g)
```

returns, but does not itself play, a random legal column.

The implementation of the method

```
String getName()
```

returns "Random Player".

Problem 3:

Write a class named YourNameAI that implements CFPlayer, where you should replace YourName with your actual name.

The implementation of the method

```
int nextMove(CFGame g)
```

returns, but does not itself play, a legal column that your AI wants to play. Your AI must be good enough to beat RandomAI 80% of the time. You do not need a sophisticated algorithm for this.

The implementation of the method

```
String getName()
```

returns "Your Name's AI".

Hint. Start by writing a very simple strategy and make it work. If you start by writing a complex strategy, you may have to spend an inordinate amount of time debugging your complex logic.

Hint. The following is a good starting point. Check if there is a winning move, and if so play it. Then check if the opponent would have a winning move, and if so block it. Otherwise play a random move by, say, calling nextMove from a RandomAI Object. Is this approach good enough? If so you're done, and if not improve upon it.

Clarification. The nextMove methods of CFPlayers don't play the moves. They just return what move they "think" you should play. The idea is to keep CFGame and CFPlayer minimal and to defer the logic of how to play the game to ConsoleCF and GUICF.

Problem 4: Write the class ConsoleCF that inherits CFGame. ConsoleCF is a command-line implementation of the Connect Four game. ConsoleCF should have the following contructors, methods, and inner class.

The constructor

```
public ConsoleCF(CFPlayer ai)
```

sets up a human vs. AI game, where the red player (the player who goes first) is randomly decided.

The constructor

```
public ConsoleCF(CFPlayer ai1, CFPlayer ai2)
```

sets up a AI vs. AI game, where the red player (the player who goes first) is randomly decided.

The method

```
public void playOut()
```

plays the game until the game is over.

The method

```
public String getWinner()
```

`return` either `"draw"`, `"Human Player"`, or the AI's name given by `CFPlayer`'s `getName` method.

The `private` inner `class`

```
private class HumanPlayer
```

implements `CFPlayer`. `HumanPlayer`'s `nextMove` implementation should print the state of the board to the command line and ask the user for the next move. If the provided move is invalid, say so and ask again for a valid move. `HumanPlayer`'s `getName` implementation should `return` `"Human Player"`.

Clarification. With this code, you should be able to run

```
public static void main(String[] args) {
  CFPlayer ai1 = new JohnAI();
  CFPlayer ai2 = new RandomAI();
  int n = 10000;
  int winCount = 0;
  for (int i=0; i<n; i++) {
    ConsoleCF game = new ConsoleCF(ai1, ai2);
    game.playOut();
    if (game.getWinner()==ai1.getName())
      winCount++;
  }
  System.out.print(ai1.getName() + " wins ");
  System.out.print(((double) winCount)/((double) n)*100 + "%");
  System.out.print(" of the time.");
}
```

Problem 5:

Write the `class GUICF` that inherits `CFGame`. `GUICF` is a graphical implementation of the Connect Four game. `GUICF` should have the following fields, contructors, methods, and inner `class`.

The `private` field

```
private GameBoard this_board;
```

represents the graphics for the 6×7 board, but not the buttons.

The constructor

```
public GUICF(CFPlayer ai)
```

sets up and starts a human vs. AI game, where the red player (the player who goes first) is randomly decided.

The constructor

```
public GUICF(CFPlayer ai1, CFPlayer ai2)
```

sets up and starts a AI vs. AI game, where the red player (the player who goes first) is randomly decided.

A human vs. AI game and an AI vs. AI game creates slightly different GUIs. A human vs. AI game will have a graphical interface like Figure 1. At each turn, the AI makes the move automatically and the human player's move is specified through clicking a button. An AI vs. AI game will have a graphical interface like Figure 2. When you click the "Play" the next AI makes the next move.

The `private` method `playGUI`

```
private boolean playGUI(int c)
```

plays the column `c`. So the internal game logic inherited from `CFGame` and the displayed board represented in `this_board` must be updated. If `c` is a column that can be played, play the column and `return true`. Otherwise `return false` and do not update the state of the game.

Write the `private class`

```
private class GameBoard extends javax.swing.JPanel {
  private GameBoard() {
    // initialize empty board
    ...
  }
  private void paint(int x, int y, int color) {
    //paints specified coordinate red or black
    ...
  }
}
```

that represents the game board. In real life, the game pieces are circular, but in `GameBoard` they can be square. These square pieces can be done with `6*7=42 JLabels` and a `GridLayout`. (You may want to use `setOpaque(...)` with the `JLabels`.) If you want circular pieces, you will have to manually draw them using the `Graphics2D` object.

The `GameBoard` does not include the buttons.

Remark. Problems 4 and 5 do not specify what happens at the end of the game. Your game should display who the winner is or that the game ended in a draw in some reasonable way.

Hint. To get a downward pointing arrow shown in Figure 1, create a `JButton` with text `"\u2193"`. The following link exaplains why this works.
`http://www.fileformat.info/info/unicode/char/2193/index.htm`

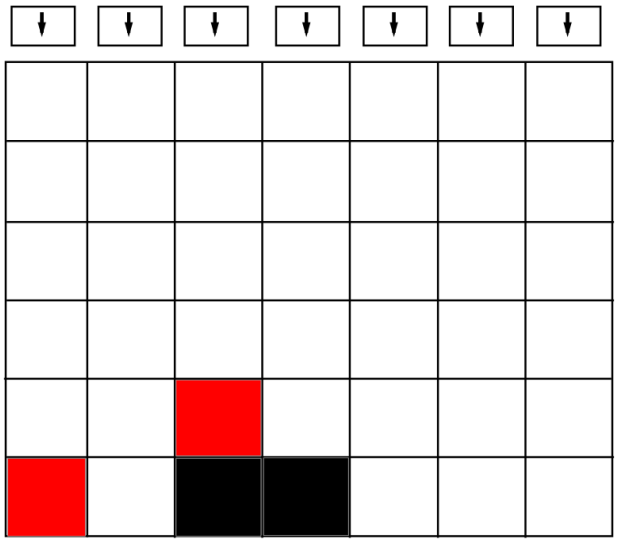

Figure 1: Human vs. AI GUI sketch

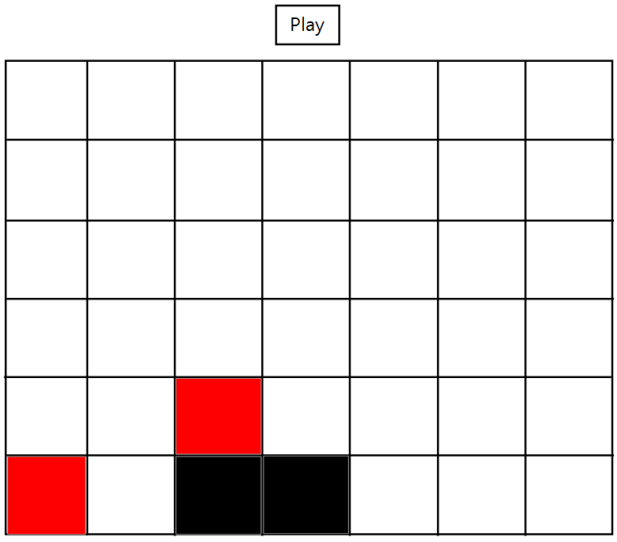

Figure 2: AI vs. AI GUI sketch