```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package fourplay;
import java.awt.Dimension;
import java.io.*;
/**
 *
 * @author Lee Hudson 09092543
 */
public class FourPlay {

    /**
     * @param args the command line arguments
     */

    public static void main(String[] args) {

        javax.swing.SwingUtilities.invokeLater(
            new Runnable() {
                public void run () {createAndShowGUI();}
            }
        );


    }
    public static void createAndShowGUI() {
        Console c = System.console();

        FPModel gameModel = new FPModel();
        FPController gameController = new FPController(gameModel);
        FPView gameView = new FPView(gameController,gameModel);
        FPScoreBoard scoreBoard = new FPScoreBoard(gameModel);
    }
}
```

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package fourplay;

import javax.swing.Timer;
import java.util.Observer;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
 *
 * @author User
 */
public class FPController {

    private FPModel gameModel;
    private FPView gameView;
    private CPUPlayer cpuOpponent;
    int startingPlayer,winningPlayer,currentPlayer,cpuPlayer;
    boolean cpuActivated;


    public FPController(FPModel gameModel){
        startingPlayer = 1;
        winningPlayer = 0;
        currentPlayer = 1;
        cpuPlayer = 1;
        cpuActivated = false;

        this.gameModel = gameModel;
        cpuOpponent = new CPUPlayer(gameModel,this,cpuPlayer);

    }

    public void setView(FPView gameView){
        this.gameView = gameView;
    }
    /**
     * This method processes a move from a human player and will be bypassed
     * if the current player is the CPU. The CPU only toggles the player variable
     * once it has completed a move and hence locks this method out during a CPU move.
     *
     * @param y         Y coordinate clicked
     * @param colomnNo  Colomn clicked on
     */
    public void mouseClickedOnPiece(int y, int colomnNo){

        if((cpuActivated && currentPlayer != cpuPlayer)||(!cpuActivated)){
            if(winningPlayer==0){
                int colomnWidth,row,rowHeight;
                int[][] boardStatus = gameModel.getChipStatus();
```

```
                colomnWidth = gameView.getBoardSize().width/gameView.getNoOfCols();
                rowHeight = gameView.getBoardSize().height/gameView.getNoOfRows();
                row = (gameView.getNoOfRows()-1)-((int)y/rowHeight);

                //Checks if the move is valid and if it is performs is.
                if(gameModel.validMove(row, colomnNo)){
                    gameModel.setPiece(colomnNo, currentPlayer);
                    togglePlayer();
                    checkForWinner();

                    //Set by checkForWinner()
                    if(winningPlayer == 0){
                        if((cpuActivated) &&(currentPlayer==cpuPlayer)){
                            cpuOpponent.doMove();
                        }
                    }
                }
            }
        }
    }
    /**
     * Called by FPView when the end game button is pressed. This
     * method clears the board and swaps the starting player.
     */
    public void endGame(){
        gameModel.clearBoard();
        if(winningPlayer > 0){
            winningPlayer = 0;
        }
        if(startingPlayer ==  1)
            startingPlayer = 2;
        else
            startingPlayer = 1;
        currentPlayer=startingPlayer;
        if(cpuActivated){
            if(currentPlayer==cpuPlayer){
                cpuOpponent.doMove();
            }
        }
    }

    /**
     * Called by FPView to reset the scores when the reset button
     * is pressed.
     */
    public void resetScores(){
        gameModel.setScore(1, 0);
        gameModel.setScore(2, 0);
    }

    /**
     * Called by the FPView to activate the CPU Player
     */
    public void setCPU(){
        if(cpuActivated){
            cpuActivated =  false;
        }else{
```

```
            cpuActivated  =  true;
            if(currentPlayer == 1){
                cpuOpponent.doMove();
            }
        }
    }

    /**
     * @return  cpuActivated
     */
    public boolean getCPU(){
        return cpuActivated;
    }

    /**
     * This method toggles the current player
     */
    public void togglePlayer(){
        if(currentPlayer == 1){
            currentPlayer = 2;
        }else{
            currentPlayer = 1;
        }
    }

    /**
     * This method checks for a winner using the method in the model winningLine()
     * and sets the winningPlayer variable accordingly. Also activates a dialogue.
     */
    public void checkForWinner(){
        if(gameModel.winningLine(1)){
            winningPlayer = 1;
            gameView.winningPlayerDialog(winningPlayer);
            gameModel.setScore(winningPlayer, gameModel.getScore(winningPlayer)+1);
        }
        if(gameModel.winningLine(2)){
            winningPlayer = 2;
            gameView.winningPlayerDialog(winningPlayer);
            gameModel.setScore(winningPlayer, gameModel.getScore(winningPlayer)+1);
        }

    }

}
```

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package fourplay;

import javax.swing.Timer;
import java.util.Observer;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Random;
/**
 *
 * @author User
 */
public class CPUPlayer implements ActionListener {

    private int playerNumber;
    private FPModel gameModel;
    private FPController gameController;
    private Timer moveTimer;

    private static final int TIME_INTERVAL = 500;

    public CPUPlayer(FPModel gameModel,FPController gameController, int playerNumber){
        this.gameModel=gameModel;
        this.gameController=gameController;
        this.playerNumber=playerNumber;

        moveTimer = new Timer(TIME_INTERVAL,this);
        moveTimer.setInitialDelay(TIME_INTERVAL);

    }

    public void setController(FPController gameController){
        this.gameController=gameController;
    }

    /**
     * This method starts a timer which once complete calls
     * requestMove().
     */
    public void doMove(){
        moveTimer.start();
    }

    /**
     * This method is call to actually perform a move. If the
     * board is empty it sets a random piece. If the board is not empty
     * it calls the bestMove() method to determine what the best move would
     * be. It then performs this move.
     */
    private void requestMove(){
```

```
        //If board is empty place a random piece
        if(gameModel.boardIsEmpty()){
            Random randomGenerator = new Random();
            int randomCol = randomGenerator.nextInt(gameModel.getNoOfColomns()-1);
            gameModel.setPiece(randomCol, playerNumber);
            gameController.togglePlayer();
            return;
        }

        //Determine which colomn click would produce the best move
        int bestMove = bestMove(playerNumber);
        if(bestMove>=0){
            gameModel.setPiece(bestMove, playerNumber);
        }
        gameController.checkForWinner();
        gameController.togglePlayer();
    }


    /**
     * This method determines which colomn click would result
     * in the best move. It does this by determining the heuristic
     * result for each possible move and picking the move with the
     * highest heuristic value.
     *
     * @param player    The player to determine the best move for.
     * @return          The best colomn to click on (0-gameModel.getNoOfColomns
     */
    public int bestMove(int player){
        int bestSoFar=0;
        int bestCol=-1;
        int currentValue;
        //Check the heuristic value of each possible move
        for(int i=0; i <gameModel.getNoOfColomns();i++){
            if(gameModel.validMove(gameModel.getNoOfRows()-1,i)){
                currentValue=heuristicResult(i,player);
                if(currentValue >=bestSoFar){
                    bestSoFar=currentValue;
                    bestCol=i;
                }
            }
        }
        return bestCol;
    }

    /**
     * Determines the value of a given move for a given player
     *
     * @param col       The colomn the piece would fall in
     * @param player    The player the piece belongs to
     * @return          A value depending of the actual value of a move,
     *                  the larger the value the better.
     */
    public int  heuristicResult(int col,int player){
        int row=0;
        int orthoResult,diagResult;
        int totalResult = 0;
```

```
        //Copys board status to do hypothetical moves
        int[][]hypoBoardStatus = new int[gameModel.getNoOfRows()][gameModel.getNoOfColomns()];
        for(int i = 0; i < gameModel.getNoOfRows(); i++){
            System.arraycopy(gameModel.getChipStatus()[i], 0, hypoBoardStatus[i], 0,
            gameModel.getNoOfColomns());
        }

        //Decides where the hypothetical chip would land for a give colomn so derives a row.
        //Also actually places the piece on hypoBoardStatus[][]
        for(int k =0 ; k < gameModel.getNoOfRows(); k++){
            if(hypoBoardStatus[k][col]==0){
                hypoBoardStatus[k][col]=player;
                row=k;
                break;
            }
        }
        //Checks the value of that move in terms of orthogonal lines
        orthoResult = heuristicOrtho(hypoBoardStatus,false,row,col,player)+heuristicOrtho(
        hypoBoardStatus,true,row,col,player);
        //Checks the value of that move in terms of diagonal lines
        diagResult = heuristicDiag(hypoBoardStatus,true,row,col,player)+heuristicDiag(
        hypoBoardStatus,false,row,col,player);

        totalResult = orthoResult+diagResult;
        return totalResult;
    }

    /**
     * This method determine the value in terms of game play for a given move. It splits
     * a row or colomn into blocks of 4 and determines the value of each block of 4. It
     * then sums the blocks together. The value of each block is determined by the method
     * blockScore().
     *
     * @param hypoBoardStatus   An int[][] representing the current state of the board
     *                          + the hypothetical move.
     * @param horizontal        Boolean, if true the method works in rows else colomns
     * @param row               Row of the hypothetical piece
     * @param col               Col of the hypothetical piece
     * @param player            Player of the hypothetical move
     * @return                  Value of the hypothetical move
     */
    public int heuristicOrtho(int[][] hypoBoardStatus, boolean horizontal,int row,int col,int
     player){

        int opponentPlayer;
        int noOfBlocks;

        if(horizontal)
            noOfBlocks=(gameModel.getNoOfColomns()-4)+1;
        else
            noOfBlocks=(gameModel.getNoOfRows()-4)+1;

        int[][] blocks = new int[noOfBlocks][4];
        int result=0;

        if(player==1){
```

```
            opponentPlayer=2;
        }else{
            opponentPlayer=1;
        }

        for(int i = 0; i < noOfBlocks; i++){
            for(int j =0; j<4;j++){
                if(horizontal){

                    blocks[i][j]=hypoBoardStatus[row][i+j];
                }else{
                    blocks[i][j]=hypoBoardStatus[i+j][col];
                }
            }
        }

        for(int k = 0; k < noOfBlocks; k++){
            if(containsOpponent(blocks[k],opponentPlayer))
                result=result+0;
            else
                result=result+blockScore(blocks[k],player);
        }
        return result;
    }

    /**
     * This method determine the value in terms of game play for a given move. It splits
     * a diagonal line into blocks of 4 and determines the value of each block of 4. It
     * then sums the blocks together. The value of each block is determined by the method
     * blockScore().
     *
     * @param hypoBoardStatus   An int[][] representing the current state of the board
     *                          + the hypothetical move.
     * @param negGrad           Boolean, true to check for negative gradient diagonals,
     *                          false to check for positive gradient diagonals.
     * @param row               Row of the hypothetical piece
     * @param col               Col of the hypothetical piece
     * @param player            Player of the hypothetical move
     * @return                  Value of the hypothetical move
     */
    public int heuristicDiag(int[][] hypoBoardStatus,boolean negGrad,int row,int col,int
    player){

        int firstDiagCoordRow, firstDiagCoordCol,lastDiagCoordRow,lastDiagCoordCol;
        int rowRes,colRes,length,opponentPlayer;
        int result = 0;

        if(player==1){
            opponentPlayer=2;
        }else{
            opponentPlayer=1;
        }

        if(negGrad){
            rowRes=gameModel.getNoOfRows()-(row+1);
            if(rowRes < col){
                firstDiagCoordRow=row+rowRes;
```

```
                firstDiagCoordCol=col-rowRes;
            }else{
                firstDiagCoordRow=row+col;
                firstDiagCoordCol=col-col;
            }

            colRes=gameModel.getNoOfColomns()-(firstDiagCoordCol+1);
            if(colRes < firstDiagCoordRow){
                lastDiagCoordRow=firstDiagCoordRow-colRes;
                lastDiagCoordCol=firstDiagCoordCol+colRes;
            }else{
                lastDiagCoordRow=firstDiagCoordRow-firstDiagCoordRow;
                lastDiagCoordCol=firstDiagCoordCol+firstDiagCoordRow;
            }
        }else{

            if(col < row){
                firstDiagCoordRow=row-col;
                firstDiagCoordCol=col-col;
            }else{
                firstDiagCoordRow=row-row;
                firstDiagCoordCol=col-row;
            }
            colRes=gameModel.getNoOfColomns()-(col+1);
            rowRes=gameModel.getNoOfRows()-(row+1);
            if(colRes < rowRes){
                lastDiagCoordRow=row+colRes;
                lastDiagCoordCol=col+colRes;
            }else{
                lastDiagCoordRow=row+rowRes;
                lastDiagCoordCol=col+rowRes;
            }

        }
        if(negGrad)
            length = (firstDiagCoordRow-lastDiagCoordRow)+1;
        else
            length = (lastDiagCoordRow-firstDiagCoordRow)+1;

        int noOfBlocks = length-3;
        if(noOfBlocks < 0)
            noOfBlocks=0;

        int[][] blocks = new int[noOfBlocks][4];

        for(int i = 0; i < noOfBlocks; i++){
            for(int j =0; j<4;j++){
                if(negGrad){
                    blocks[i][j]=hypoBoardStatus[firstDiagCoordRow-(j+i)][firstDiagCoordCol+(
                    j+i)];
                }else{
                    blocks[i][j]=hypoBoardStatus[firstDiagCoordRow+(j+i)][firstDiagCoordCol+(
                    j+i)];
                }
            }
        }
        for(int k = 0; k < noOfBlocks; k++){
```

```
            if(containsOpponent(blocks[k],opponentPlayer))
                result=result+0;
            else
                result=result+blockScore(blocks[k],player);
        }
        return result;
    }
    /**
     * This method returns a score for a block of pieces where:
     * 1 piece = score of 1
     * 2 consecutive pieces = score of 4
     * 3 consecutive pieces = score of 32
     * 4 consecutive pieces = score of 128 (this is a win)
     *
     * @param subject   The block to examine
     * @param player    The player in question
     * @return          The total score for the block
     */
    public int blockScore(int[] subject, int player){
        int counter=0;
        int result=0;

        for(int i = 0; i < subject.length; i++){
            if(subject[i]==player){
                switch (counter){
                    case 0: counter++;
                        break;
                    case 1: counter = counter +3;
                        break;
                    case 4: counter = counter +28;
                        break;
                    case 32: counter = counter +96;
                        break;
                    default: counter = -999;
                        break;
                }

            }else{
                result = result+counter;
                counter = 0;
            }
        }
        return result+counter;
    }


    /**
     * This method checks a block of pieces and if there is a piece
     * from the opponent it returns true, else false.
     *
     * @param subject   The block to examine
     * @param opponent  The opponent player
     * @return          True if the opponents chip is found
     */
    public boolean containsOpponent(int[] subject, int opponent){
        for(int i= 0; i < subject.length; i++){
            if(subject[i]== opponent)
```

```
                return true;
            }
            return false;
        }

        public void actionPerformed(ActionEvent event){
            if(event.getSource()==moveTimer){
                requestMove();
                moveTimer.stop();
            }
        }
}
```

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package fourplay;

import java.util.Observer;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
/**
 *
 * @author Lee Hudson   09092543
 * This class is the main board GUI. It contains many ColomnPanel objects
 * to show the game pieces.
 */
public class FPView implements Observer, ActionListener {

    private Dimension boardSize;
    private int noOfColomns;
    private int noOfRows;
    private JFrame board;
    private JPanel boardPanel,buttonPanel;
    private JButton reset,endGame,cpu;
    private ColomnPanel[] colomns;
    private JLabel player1Score,player2Score,player1Label,player2Label;
    private FPController gameController;
    private FPModel gameModel;

    public FPView(FPController gameController, FPModel gameModel){
        this.gameController=gameController;
        this.gameModel=gameModel;

        noOfColomns=gameModel.getNoOfColomns();
        noOfRows=gameModel.getNoOfRows();
        boardSize = new Dimension(noOfColomns*100,noOfRows*100);

        gameModel.addObserver(this);
        createBoard();
        gameController.setView(this);
    }

    /**
     * Sets up the GUI components
     */
    public void createBoard(){

        //Initialise GUI components
        board = new JFrame("FourPlay");
        boardPanel = new JPanel();
        buttonPanel = new JPanel();
        reset = new JButton("Reset");
```

```
        endGame = new JButton("End Game");
        cpu = new JButton("Activate CPU");
        colomns = new ColomnPanel[noOfColomns];

        //Set action listners for the buttons
        reset.addActionListener(this);
        endGame.addActionListener(this);
        cpu.addActionListener(this);

        //Set up main JFrame"board"
        board.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = board.getContentPane();
        contentPane.setLayout(new BoxLayout(contentPane, BoxLayout.Y_AXIS));

        //Set up JPanel that contains the chips "boardPanel". The board panel
        //is made up of ColomnPanel objects.
        boardPanel.setLayout(new BoxLayout(boardPanel, BoxLayout.X_AXIS));
        for(int i=0; i < noOfColomns; i++ ){
            colomns[i]= new ColomnPanel(boardSize.width/noOfColomns,boardSize.height,noOfRows
            ,i,gameModel,this);
            boardPanel.add(colomns[i]);
        }

        //Set up the button JPanel "buttonPanel"
        buttonPanel.setLayout(new GridLayout(1,2));
        buttonPanel.add(reset);
        buttonPanel.add(endGame);
        buttonPanel.add(cpu);
        reset.setEnabled(false);
        endGame.setEnabled(false);

        //Add components to the main JFrame
        contentPane.add(boardPanel);
        contentPane.add(buttonPanel);

        board.pack();
        board.setResizable(false);
        board.setVisible(true);

    }

    /**
     *
     * @return NO_OF_COLOMNS
     */
    public int getNoOfCols(){
        return noOfColomns;
    }

    /**
     *
     * @return NO_OF_ROWS
     */
    public int getNoOfRows(){
        return noOfRows;
    }
```

```
    /**
     *
     * @return BOARD_SIZE
     */
    public Dimension getBoardSize(){
        return boardSize;
    }

    public void update(java.util.Observable o, Object arg){
        if(gameModel.boardIsEmpty()){
            if(reset.isEnabled()){
                reset.setEnabled(false);
                endGame.setEnabled(false);
            }
        }else{
            if(reset.isEnabled()==false){
                reset.setEnabled(true);
                endGame.setEnabled(true);
            }
        }
        board.repaint();

    }

    /**
     * Called by the colomnPanel class when mouse is clicked on a colomn. Passes
     * click details to gameController.mouseClickedOnPiece()
     *
     * @param y         The Y coordinate of the click
     * @param colomnNo  The number of the colomnPanel that called
     *                  the method and hence the colomn.
     */
    public void mouseClicked(int y, int colomnNo){
        gameController.mouseClickedOnPiece(y, colomnNo);

    }


    /**
     * Displays a dialogue when a player wins
     * @param player    Player number of the winning player
     */
    public void winningPlayerDialog(int player){
        JOptionPane.showMessageDialog(board, "Player "+player+" wins!");
    }

    public void actionPerformed(ActionEvent event){
        if(event.getSource()==endGame){
            gameController.endGame();
        }
        if(event.getSource()==reset){
            gameController.resetScores();
        }
        if(event.getSource()==cpu){
            gameController.setCPU();

            if(gameController.getCPU()){
```

```
                cpu.setText("De-activate CPU");
            }else{
                cpu.setText("Activate CPU");
            }

        }

    }

}
```

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package fourplay;
import javax.swing.*;
import java.awt.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

/**
 *
 * @author Lee Hudson 09092543
 * This class is a specialised JPanel that displays a single colomn of
 * the board. It handles all clicking events associated with it.
 *
 */
public class ColomnPanel extends JPanel implements MouseListener {

    private Dimension size;
    private int colomnNumber;
    private int chipWidth,chipHeight,verticalSpacing,x,y,noOfChips;
    FPModel model;
    FPView gameView;


    public ColomnPanel(int width, int height, int noOfChips, int colomnNumber, FPModel model,
    FPView gameView){

        this.size = new Dimension(width,height);
        chipWidth=(int)(width-(width*0.1));
        chipHeight=(int)(height/noOfChips-((height/noOfChips)*0.1));
        x=(size.width-chipWidth)/2;
        this.noOfChips=noOfChips;
        y=(height/noOfChips)-chipHeight;
        this.colomnNumber=colomnNumber;
        this.model = model;
        this.gameView=gameView;
        this.addMouseListener(this);
    }


    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int[][] boardStatus = model.getChipStatus();
        for(int i=0; i < noOfChips; i++){
            switch(boardStatus[(noOfChips-1)-i][colomnNumber]){
                case 0:
                    g.setColor(Color.GRAY);
                    break;
                case 1:
                    g.setColor(Color.RED);
                    break;
                case 2:
                    g.setColor(Color.BLACK);
```

```
                    break;
                default:
                    g.setColor(Color.GRAY);
                    break;
            }
            g.fillOval(x,((chipHeight+y)*i),chipWidth,chipHeight);
        }
    }

    /**
     *
     * @return colomnNumber
     */
    public int getColomnNumber(){
        return colomnNumber;
    }

    @Override
    public Dimension getPreferredSize(){
        return size;
    }

    @Override
    public Dimension getMinimumSize(){
        return getPreferredSize();
    }

    @Override
    public Dimension getMaximumSize(){
        return getPreferredSize();
    }


    public void mouseClicked(MouseEvent e) {
    }

    public void mousePressed(MouseEvent e) {
        gameView.mouseClicked(e.getY(), colomnNumber);
    }
    public void mouseReleased(MouseEvent e) {
    }
    public void mouseEntered(MouseEvent e) {
    }
    public void mouseExited(MouseEvent e) {
    }

    public void mouseMoved(MouseEvent e) {

    }

    public void mouseDragged(MouseEvent e) {
    }

}
```

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package fourplay;

/**
 *
 * @author User
 */
import java.util.Observer;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
/**
 *
 * @author User
 */
public class FPScoreBoard implements Observer {


    private JFrame scoreBoard;
    private JPanel scorePanel;
    private JLabel player1Score,player2Score,player1Label,player2Label;
    private FPModel gameModel;

    public FPScoreBoard(FPModel gameModel){
        this.gameModel=gameModel;

        gameModel.addObserver(this);
        createBoard();
    }

    public void createBoard(){

        //Initialise GUI components
        //scoreBoard = new JFrame("ScoreBoard");
        scorePanel = new JPanel();

        player1Label = new JLabel("", JLabel.CENTER);
        player2Label = new JLabel("",JLabel.CENTER);
        player1Score = new JLabel("", JLabel.CENTER);
        player2Score = new JLabel("",JLabel.CENTER);

        player1Label.setText("Player 1");
        player2Label.setText("Player 2");
        player1Score.setText(""+gameModel.getScore(1));
        player2Score.setText(""+gameModel.getScore(2));

        scoreBoard = new JFrame("Java Swing Examples");
        scoreBoard.setPreferredSize(new Dimension(200,200));
        scoreBoard.setLayout(new GridLayout(2, 2));
```

```
        scoreBoard.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent windowEvent){
                System.exit(0);
         }
        });

        scoreBoard.add(player1Label);
        scoreBoard.add(player2Label);
        scoreBoard.add(player1Score);
        scoreBoard.add(player2Score);

        scoreBoard.pack();
        scoreBoard.setResizable(false);
        scoreBoard.setVisible(true);

    }

    public void update(java.util.Observable o, Object arg){
        player1Score.setText(""+gameModel.getScore(1));
        player2Score.setText(""+gameModel.getScore(2));
        scoreBoard.repaint();
    }


}
```

```
package fourplay;
import java.util.Observable;

/**
 *
 * @author Lee Hudson 09092543
 */
public class FPModel extends Observable {

    private /*@ spec_public @*/ final int noOfColomns = 7;
    private /*@ spec_public @*/ final int noOfRows = 6;
    private /*@ spec_public @*/ int[][] boardStatus;
    private /*@ spec_public @*/ int player1,player2;
    private /*@ spec_public @*/ boolean boardEmpty = true;


    public FPModel(){
        player1 = 0;
        player2 = 0;
    boardStatus = new int[noOfRows][noOfColomns];
        clearBoard();
    }

    /**
     * This method clears the board and sets the boardEmpty variable to true
     *
     * @ invariant boardStatus.length == noOfRows;
     * @ invariant (\forall int i; 0 <= i && i < noOfRows;
     *        boardStatus[i].length == noOfColomns);
     * @ ensures (\forall int j,k; 0 <= j && j < noOfRows && 0 <= k && k < noOfCololmns;
     *      boardStatus[j][k] == 0;
     * @ ensures boardEmpty == true;
     */
    public void clearBoard(){

        for(int i = 0; i < noOfRows; i++){
            for(int j = 0; j < noOfColomns; j++){
                boardStatus[i][j] = 0;
            }
        }
        boardEmpty = true;
        setChanged();
        notifyObservers();
    }


    /**
     * Getter methods for the noOfColomns
     *
     * @return      The variable noOfColoms
     *
     * @ ensures \result == noOfColomns
     */
    public int getNoOfColomns() {
        return noOfColomns;
    }
```

```
    /**
     * Getter method for the noOfRows variable
     *
     * @return      The variable noOfRows
     *
     * @ ensures \result == noOfRows
     */
    public int getNoOfRows() {
        return noOfRows;
    }

    /**
     * Returns the status of the board
     *
     * @return      The variable boardStatus
     *
     * @ ensures \result == boardStatus
     */
    public int[][] getChipStatus() {
        return boardStatus;
    }

    /**
     * Getter method to retrieve the score for a given player
     *
     * @param player    The player to get the score for
     * @return          The score of the given player
     *
     * @ requires player == 1 || player == 2;
     * @ ensures player == 1 ==> (\result == player1);
     * @ ensures player == 2 ==> (\result == player2);
     */
    public int getScore(int player){
        switch (player){
            case 1: return player1;
            case 2: return player2;
            default: return 0;
        }
    }

    /**
     * This method sets a given score for a given player
     *
     * @param player    The player to set the score for
     * @param score     The new score that is to be set
     *
     * @ requires player == 1 || player == 2;
     * @ ensures player == 1 ==> (player1 == score);
     * @ ensures player == 2 ==> (player2 == score);
     */
    public void setScore(int player, int score){
        if(player == 1)
            player1=score;
        if(player == 2)
            player2=score;
        setChanged();
```

```
        notifyObservers();
    }

    /**
     * This method informs the caller if a given move is valid.ie. if the
     * position that is clicked is free.
     *
     * @param row   Row that was clicked
     * @param col   Colomn that was clicked
     * @return      True if position is empty, false if not.
     *
     * @ requires row < noOfRows && row >= 0;
     * @ requires colomn < noOfColomns && colomn >=0;
     * @ ensures \result == true ==> boardStatus[row][col]==0;
     */
    public boolean validMove(int row, int col){
        if(boardStatus[row][col]==0)
            return true;
        else
            return false;

    }

    /**
     * Sets the piece situated at above the lowest piece in a given colomn
     * and sets the boardEmpty variable to signify that the board is no
     * longer empty.
     *
     * @param colomn    The colomn where the player clicked
     * @param value     The player that clicked
     *
     * @ requires colomn < noOfColomns && colomn >=0;
     * @ ensures  (\forall in i; 0 <= i && i < noOfRows;
                    \old boardStatus[i][colomn] == 0 => boardStatus[i][colomn]=value;
     * @ ensures boardEmpty == true ==> (boardEmpty == false);
     * @ invariant (\forall int j,k; 0 <= j && j < noOfRows &&
     *              0 <= k && k < noOfCololmns;
     *              boardStatus[j][k] != 0 => boardStatus[j][k] == \old boardStatus[j][k];
     */
    public void setPiece(int colomn, int value){
        if(boardEmpty==true)
            boardEmpty=false;

        for(int i =0 ; i < noOfRows; i++){
            if(boardStatus[i][colomn]==0){
                boardStatus[i][colomn]=value;
                break;
            }
        }
        setChanged();
        notifyObservers();
    }

    /**
     * Getter method for the variable boardEmpty
     *
     * @return      boardEmpty
```

```
     *
     * @ ensures \result == boardEmpty;;
     */
    public boolean boardIsEmpty(){
        return boardEmpty;
    }

    /**
     * This method traverses the boardStatus structure to determine
     * if there is a line of 4 or more pieces of any given color
     * in either a horizontal,vertical or diagonal orientation.
     *
     * @param player The player number of the player to check for
     * @return       True if winning line is found, false if not
     */
    public boolean winningLine(int player){
        for(int row = 0; row < noOfRows;row++){
            for(int col = 0; col < noOfColomns; col++){
                if(hasNeighbour(1,1,row,col,player) >=4)
                    return true;
                if(hasNeighbour(1,0,row,col,player) >=4)
                    return true;
                if(hasNeighbour(0,1,row,col,player) >=4)
                    return true;
                if(hasNeighbour(1,-1,row,col,player) >=4)
                    return true;
            }
        }
        return false;
    }

    /**
     * This method checks if there is a piece belonging to a given player
     * int the coordinates row and col. If there is it recursively calls
     * itself to check on the next position along, adding up the consecutive
     * pieces as it goes along. Once it finds a piece that does not belong
     * to the given player it breaks the recursion with a return 0.
     *
     * @param xDir  Determines the direction of the search in the x orientation
     *              0 is no search, 1 is positive and -1 is negative.
     * @param yDir  Determines the direction of the search in the y orientation
     *              0 is no search, 1 is positive and -1 is negative.
     * @param row   Determines the row to check
     * @param col   Determines the colomn to check
     * @param player Determines the player to check for.
     * @return      Number of consecutive pieces found for the given player.
     */
    private int hasNeighbour(int xDir,int yDir,int row, int col, int player){
        int found=0;
        if((row>=noOfRows||row<0)||(col>=noOfColomns||col<0))
            return 0;
        if(boardStatus[row][col]==player){
            found=1;
            if((xDir == 1)&&(yDir == 1)){
                //Up diagonal search
                return found+hasNeighbour(xDir,yDir,row+1,col+1,player);
            }
```

```
            if((xDir == 1)&&(yDir == 0)){
                //Horizontal search
                return found+hasNeighbour(xDir,yDir,row,col+1,player);
            }
            if((xDir == 0)&&(yDir == 1)){
                //Vertical search
                return found+hasNeighbour(xDir,yDir,row+1,col,player);
            }
            if((xDir == 1)&&(yDir == -1)){
                //Down diagonal search
                return found+hasNeighbour(xDir,yDir,row-1,col+1,player);
            }
            return 0;
        }else{
            return 0;
        }
    }


}
```