

```

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    Column that was clicked
 * @return       True if position is empty, false if not.
 *
 * @ requires row < noOfRows && row >= 0;
 * @ requires column < noOfColumns && column >=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 column
 * and sets the boardEmpty variable to signify that the board is no
 * longer empty.
 *
 * @param column The column where the player clicked
 * @param value   The player that clicked
 *
 * @ requires column < noOfColumns && column >=0;
 * @ ensures (\forall i in i; 0 <= i && i < noOfRows;
 *           \old boardStatus[i][column] == 0 => boardStatus[i][column]=value;
 * @ ensures boardEmpty == true ==> (boardEmpty == false);
 * @ invariant (\forall int j,k; 0 <= j && j < noOfRows &&
 *           0 <= k && k < noOfColumns;
 *           boardStatus[j][k] != 0 => boardStatus[j][k] == \old boardStatus[j][k];
 */
public void setPiece(int column, int value){
    if(boardEmpty==true)
        boardEmpty=false;

    for(int i =0 ; i < noOfRows; i++){
        if(boardStatus[i][column]==0){
            boardStatus[i][column]=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 < noOfColumns; 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 column 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>=noOfColumns||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;
    }
}

}
```