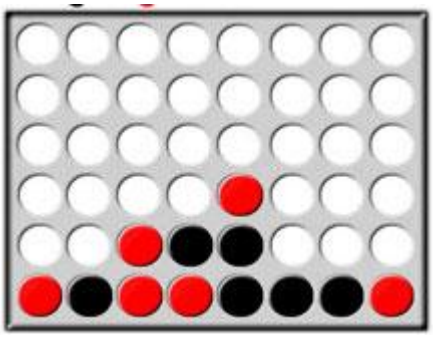
AI 06-07
Implementation of Connect4

Connect Four

Connect-4 is a two player game which is played on a 7x6 rectangular board placed vertically between them. Each player can drop a token at the top of the board in any one of the seven columns, as long as this column is not full (6 tokens in one column). The objective of the game is to align four connected (neighbouring) tokens of the same colour either horizontally, vertically or diagonally. The game ends in a draw if the board is full and all 42 tokens have been played without either player achieving four connected tokens. Players alternate and are forced to move, rule which becomes particular relevant towards the end of the game as more and more columns fill up.



The exercises below ask you to implement a Java program able to play the game against a human opponent. For example where a person plays red and the computer plays black. An input parameter to the program will specify a maximum depth cutoff for black's mini-max game tree search. For example if you run the game with a depth cut-off of 4, black may only build the game tree 4-ply deep.

Exercise 1

Encode the game in Java as an adversarial search problem.

Exercise 2

Implement the alpha-beta pruning search algorithm with a cutoff search limit. The chosen cutoff limit should be one of the parameters of your program. Because of the cutoff limit you should define an evaluation function. Invent one of your own, or search on the web for one you think can be appropriate. Explain the evaluation function rational.

Exercise 3

Use the outcomes of Exercise 1 and 2 to implement a game playing program. The computer should be able to play both as first or second player. At each turn the program should first print the current configuration of the board, and decide and play its move, or ask the opponent an input to indicate the column where the coin should be inserted. The loop should continue, alternating the turns until the game is over (by either draw or one of the players' win). The board state should be printed by using 'X' and 'O' to indicate the different coins.

The game-board

The game-field is represented by a two-dimensional array

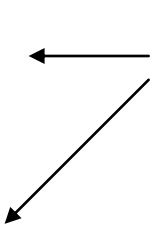
```
private Cell[][] grid = new Cell[HEIGHT][WIDTH];
```

of **Cell** objects. The class Cell is the abstraction of one cell of the game-board and contains if it is occupied and if, which player lies on it.

```
public class Cell {
	private boolean isOccupied;
	private int player; //1 --> 1st player, 2 --> 2nd player(CPU)

	public Cell(){
		isOccupied = false;
		player = -1;
	}
	public void placeCoin(int playerNum){
		isOccupied = true;
		player = playerNum;
	}

	public void removeCoin(){
		player = -1;
		isOccupied = false;
	}

	public int getPlayer(){
		return player;
	}
	public String toString(){
		String tmp;
		if(!isOccupied){
			tmp = "[ ]";
		}else{
			if(player==1)
				tmp = "[O]";
			else
				tmp = "[X]";
		}
		return tmp;
	}
}
```

The class GameField holds the two-dimensional game-board and information variables such as the width and height of the board and an array that indicates the height of each column on the board in order to know if it is possible to insert another coin on that column.

```
 1  2  3  4  5  6  7
[ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][O][ ][ ][ ]
[X][ ][ ][O][ ][X][ ]
 1  2  3  4  5  6  7

[1, 0, 0, 2, 0, 1, 0]
```



Game board and columnHeight-array at some state of the game.

```
public class GameField {
        public final int HEIGHT = 6;
        public final int WIDTH = 7;
        public int[] columnHeight;
        private Cell[][] grid = new Cell[HEIGHT][WIDTH];
        ...
        ...
}
```

Moreover it has methods such as getting the current player on one of the cells, inserting and removing coins from a column, where the removing method is used by the min-max algorithm.

```
        public void placeCoin(int column, int player){
                grid[columnHeight[column]][column].placeCoin(player);
                columnHeight[column]++;
        }

        public void removeCoin(int column){
                columnHeight[column]--;
                grid[columnHeight[column]][column].removeCoin();
        }

        public int getPlayer(int column, int row){
                return grid[row][column].getPlayer();
        }
```

The Game controller

The GameController is the class that controls the flow of the game. It creates the game-board by creating an object by invoking the class **`GameField`**, and contains also the core methods of the game such as the min-max algorithm and the **`checkFour()`** method which checks for the termination of the game.
Other hold information is the maximum depth of the recursion (cutoff for the min-max algorithm), the current recursive depth (used in the min-max algorithm) and a String array holding simply the names of the opponents (if two human players).

```
public class GameController {
        private GameField field;
        private int MAX_RECURDEPTH = 4;
        private int recurDepth=0;
        private String[] opponents;

        public GameController() throws IOException{
                opponents = new String[2];
                field = new GameField();
        }
        ...
        ...
}
```

The “cycling” of the game takes place in the **`playCPU()`** method. This method

- reads the input from the human player,
- makes decisions if it’s the CPU’s turn,
- checks for a winner and

- switches the players

after each cycle. The players are represented as Integers, where the human player has number 1 and the CPU has number 2. In this way the switching can be done by subtracting the current player from the number 3 after each cycle.

```
...
player = 3 - player;
...
```

```
public void playCPU() {
        // get CPU skill level
        MAX_RECURDEPTH = readCPUSkills();
        // read player names
        readPlayernames(true);

        // variable to store input from user
        int column;
        // player to start; is decided randomly between CPU and human
        int player = randomPlayer();

        while (true) {
                if (player == 1) {
                        //human plays here
                        System.out.println(field);
                        do {
                                column = getColumnInput(player) - 1;
                        } while (column < 0 || column >= field.WIDTH
                                        || field.columnHeight[column] >= field.HEIGHT);
                } else {
                        // cpu plays here
                        column = minmaxDecision(player);
                        if(column==-1){
                             System.out.println("Nobody has won the game! It's a draw!");
                                break;
                        }
                }
                field.placeCoin(column, player);

                int winner = 0;
                winner = checkFour();
                if (winner > 0) {
                        System.out.println(field);
                        System.out.println(opponents[winner - 1] + " has won the game!");
                        break;
                }
                System.out.println("\n");
                if(player==2)
                        System.out.println("Info: CPU has put his coin in column " +
(column+1) + "!");
                player = 3 - player;
        }
}
```

Choosen strategy

The strategy of my implementation is not to create first the tree of the different game-states and then to evaluate that tree, but to have the tree implicit in the recursion of the min-max algorithm.

I divided the min-max algorithm into two separate methods. When during the game it is the turn of the CPU, the **`minmaxDecision(…)`** is called:

```
private int minmaxDecision(int player){
        int bestCol = -1;
        ArrayList<Integer> bestCols = new ArrayList<Integer>();
        int prevBestVal = -MAX_RECURDEPTH - 1;

        for(int x=0; x<field.WIDTH; x++){
                if(field.columnHeight[x]<field.HEIGHT){
                        int val = max(player,x);
                        if(val>prevBestVal){
                                bestCols.clear();
                                prevBestVal = val;
                        }
                        if(val==prevBestVal){
                                bestCols.add(x);
                        }
                }
        }


        if(bestCols.size()>0){
                Collections.shuffle(bestCols);
                bestCol = bestCols.get(0);
        }
        return bestCol;
}
```

This method calls the **`max()`** method (since CPU wants to maximize) of the min-max algorithm for each column of the game-field, giving the **`max()`** method as parameter the player and the current column. The player remains in this case always 2 which indicates the CPU. For every outcome of the **`max()`** method with the same value as the previous, the associated column is stored in the **`ArrayList bestCols`**. If a better value is found, the whole ArrayList-content is cleared and filled up again. At the end, the ArrayList is shuffled and then the first entry is taken as the choice.

The **`max()`** method is the following:

```
private int max(int player, int column){
        int value = 0;
        if(field.columnHeight[column]>=field.HEIGHT)
                return 0;

        recurDepth++;
        field.placeCoin(column, player);

        if(checkFour()>0){
                value = MAX_RECURDEPTH+1-recurDepth;
        }

        if(recurDepth<MAX_RECURDEPTH && value==0){
                value = MAX_RECURDEPTH+1;
```

```
                for(int x=0; x<field.WIDTH; x++){
                        if(field.columnHeight[x]<field.HEIGHT){
                                int tmpVal = min(3-player,x);
                                if(value==(MAX_RECURDEPTH+1)){
                                        value = tmpVal;
                                }else if(tmpVal<value){
                                        value = tmpVal;
                                }
                        }
                }
        }

        recurDepth--;
        field.removeCoin(column);

        return value;
}
```

The steps each time the max or min-method is called are basically the following:

- value is set to 0, because it is in some sense a neutral value (middle of min and max)
- it is checked, that the chosen column is not full

```
if(field.columnHeight[column]>=field.HEIGHT)
        return 0;
```

- the recursive depth is increased
- a coin is inserted at the specified column by the specified player
- the evaluation function is called (**`checkFour()`**)
- as next step the cutoff-limit and the condition for the alpha-beta pruning is checked

```
if(recurDepth<MAX_RECURDEPTH && value==0){
        value = MAX_RECURDEPTH+1;
        for(int x=0; x<field.WIDTH; x++){
                ...
                ...
```

cutoff-limit check

check if somebody already won since then it is unnecessary to go on.

  Checking for **`value==0`** checks basically if the player has already four in the row up to now in the recursion (in the current game-state). If this is the case, it is no more needed to go deeper in the recursion since we already found the min or max, depending on the player.

- if all the above conditions hold, the algorithm goes into the recursion, by calling on each column the **`min()`** method for the opponent player and storing the minimum / maximum value which is then pushed up.

Evaluation function

The evaluation function is the part where the **`checkFour()`** is called. In the case of the **`max()`**, the **`checkFour()`** will either return 0, meaning that nobody has four in a row or it will return 2, which means that the 2nd player – which is the CPU – has won. It is not possible here that it returns 1, since the 2nd player was the

last to insert a coin and therefore only he has the chance to win, since otherwise the game would have terminated before.
If the 2nd player has four in a row, the distance to the victory is computed by subtracting the current recursive depth from the maximum recursive depth possible + 1:

```
if(checkFour()>0){
        value = MAX_RECURDEPTH+1-recurDepth;
}
```

Example of a game state:

The CPU is the player X and the current situation is the following:

```
 1  2  3  4  5  6  7
[ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][O][ ][ ]
[ ][ ][ ][ ][O][ ][ ]
[ ][ ][ ][ ][O][X][X]
 1  2  3  4  5  6  7
```

The **minmaxDecision()** method will apply the **max()** method on all the columns. With a maximum recursive depth of 4, the max() method will return the value -3 on all columns, except the column 4, where it will return 0. It returns -3 because in the 2nd depth of the recursion – in the min-phase of the opponent player – the opponent will win, by putting his coin in column 4. This means that the program enters this part:

```
if(checkFour()>0){
        value = -MAX_RECURDEPTH-1+recurDepth;
}
```

Since **MAX_RECURDEPTH** = 4 and **recurDepth** = 2, this gives -3.
Applying instead the max() method on column 4 will return 0, since in that case, the CPU would place his coin in the 1st d of the recursion on column 4, causing that the opponent player will never have the possibility to get four in a row at every state in the game-state tree. Therefore 0 is return meaning that nobody wins in that column. However, since the CPU tries to maximize, in the **minmaxDecision** method the value 0 will be recorded as the biggest value in contrast to the -3 of all the other columns. So the ArrayList **bestCols** will remove all its entries and insert column 4 as the only column. So the CPU will put his coin in column four, reaching this state of the game:

```
 1  2  3  4  5  6  7
[ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][X][ ][ ][ ]
[ ][ ][ ][O][ ][ ][ ]
[ ][ ][ ][O][ ][ ][ ]
[ ][ ][ ][O][X][X][ ]
 1  2  3  4  5  6  7
```

Alpha-beta pruning

The max-part of the alpha beta pruning looks as follows:

```
private int maxAlphaBeta(int player, int column, int alpha, int beta){
        boolean winnerFound = false;

        if(field.columnHeight[column]>=field.HEIGHT)
                return 0;

        recurDepth++;
        field.placeCoin(column, player);

        if(checkFour()>0){
                //max-player has 4 in a row
                alpha = MAX_RECURDEPTH+1-recurDepth;
                winnerFound = true;
        }
        if(recurDepth==MAX_RECURDEPTH && !winnerFound){
                //set alpha to a neutral value if lowest nodes in
                //recursion-tree are visited without a result
                alpha = 0;
        }else if(recurDepth<MAX_RECURDEPTH && !winnerFound){
                for(int i=0; i<field.WIDTH; i++){
                        if(field.columnHeight[i]<field.HEIGHT){
                                int tmpVal = minAlphaBeta(3-player, i, alpha, beta);
                                if(alpha==(-MAX_RECURDEPTH-1)){
                                        alpha = tmpVal;
                                }else if(alpha>tmpVal){
                                        alpha = tmpVal;
                                }

                                if(alpha>=beta){
                                        break;
                                }

                        }
                }
        }

        recurDepth--;
        field.removeCoin(column);
        return alpha;
}
```

This method is basically the extended method of the normal minmax. The part

```
        if(recurDepth==MAX_RECURDEPTH && !winnerFound){
                //set alpha to a neutral value if lowest nodes in
                //recursion-tree are visited without a result
                alpha = 0;
        ...
```

had to be put, since in the normal **`max(…)`** method there is the variable **`value`** which is initialized to a neutral value zero. If no winner is found in the recursion tree this value which remains zero is returned. In the alpha-beta max **`maxAlphaBeta`** this variable value has been dropped and replaced by alpha or beta. Therefore it was necessary to make sure that if the recursion has reached the bottom-most level and no winner was found, the alpha or beta is set to zero. The minmaxDecision(…) has remained the same.

Launcher.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Launcher {


	public static void main(String[] args) {
		System.out.println("	  ****************************************");
		System.out.println("	  **		JConnect 4		**");
		System.out.println("	  ****************************************\n");

		BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
		//reading play-mode
		int playmode = 0;
		do{
			System.out.print("Please choose playmode (1=against human; 2=against CPU):");
			try {
				playmode = Integer.parseInt(br.readLine());
			} catch (Exception e) {
				playmode = 0;
			}
		}while(playmode<1 || playmode>2);

		try {
			GameController controller = new GameController();
			if(playmode==1){
				controller.playHumans();
			}else{
				controller.playCPU();
			}
		} catch (IOException e) {
			e.printStackTrace();
		}

	}

}
```

GameController.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Random;
/**
 * This class is the controller of the game. It reads in the players, makes the
 * CPU's decisions...basically it controls the game-flow
 *
 * @author Juri Strumpflohner
 *
 */
public class GameController {
	private GameField field;
	private int MAX_RECURDEPTH = 4;
	private int recurDepth=0;
	private String[] opponents;

	public GameController() throws IOException{
		opponents = new String[2];
		field = new GameField();
	}

	public void playHumans(){
		readPlayernames(false);

		//variable to store input from user
		int column;
		//player to start
		int player = randomPlayer();

		while(true){
			if(player==1){
				System.out.println(field);
				do {
			column=getColumnInput(player)-1;
		} while(column<0 || column >=field.WIDTH || field.columnHeight[column]>=field.HEIGHT);
			}else{
				System.out.println(field);
				do {
			column=getColumnInput(player)-1;
		} while(column<0 || column >=field.WIDTH || field.columnHeight[column]>=field.HEIGHT);
			}
			field.placeCoin(column, player);

			int winner = 0;
			winner = checkFour();
			if(winner>0){
				System.out.println("Player " + opponents[winner] + " has won!" );
				break;
			}
			player=3-player;
		}
	}

	public void playCPU() {
		// get CPU skill level
		MAX_RECURDEPTH = readCPUSkills();
		// read player names
		readPlayernames(true);

		// variable to store input from user
		int column;
		// player to start; is decided randomly between CPU and human
		int player = randomPlayer();

		while (true) {
```

```
                        if (player == 1) {
                                //human plays here
                                System.out.println(field);
                                do {
                                        column = getColumnInput(player) - 1;
                                } while (column < 0 || column >= field.WIDTH
                                                || field.columnHeight[column] >= field.HEIGHT);
                        } else {
                                // cpu plays here
                                column = minmaxDecision(player);
                                if(column==-1){
                                        System.out.println("Nobody has won the game! It's a draw!");
                                        break;
                                }
                        }
                        field.placeCoin(column, player);

                        int winner = 0;
                        winner = checkFour();
                        if (winner > 0) {
                                System.out.println(field);
                                System.out.println(opponents[winner - 1] + " has won the game!");
                                break;
                        }
                        System.out.println("\n");
                        if(player==2)
                                System.out.println("Info: CPU has put his coin in column " + (column+1) +
"!");
                        player = 3 - player;
                }
        }

        /**
         * This method is called by the CPU to make a choice of the column. The method
         * calls the minmax algorithm, by invoking the max() method, since the CPU
         * tries to maximize.
         *
         * @param player - will be always the CPU, so player=2
         * @return an integer indicating the best column where the CPU has to
         *                      put its coin
         */
        private int minmaxDecision(int player){
                int bestCol = -1;
                ArrayList<Integer> bestCols = new ArrayList<Integer>();
                int prevBestVal = -MAX_RECURDEPTH - 1;

                for(int x=0; x<field.WIDTH; x++){
                        if(field.columnHeight[x]<field.HEIGHT){
//                              int val = max(player,x);
                                int val = maxAlphaBeta(player, x, -MAX_RECURDEPTH-1, MAX_RECURDEPTH+1);
                                if(val>prevBestVal){
                                        bestCols.clear();
                                        prevBestVal = val;
                                }
                                if(val==prevBestVal){
                                        bestCols.add(x);
                                }
                        }
                }


                if(bestCols.size()>0){
                        Collections.shuffle(bestCols);
                        bestCol = bestCols.get(0);
                }
                return bestCol;
        }

        private int maxAlphaBeta(int player, int column, int alpha, int beta){
                boolean winnerFound = false;
```

```
		if(field.columnHeight[column]>=field.HEIGHT)
			return 0;

		recurDepth++;
		field.placeCoin(column, player);

		if(checkFour()>0){
			//max-player has 4 in a row
			alpha = MAX_RECURDEPTH+1-recurDepth;
			winnerFound = true;
		}
		if(recurDepth==MAX_RECURDEPTH && !winnerFound){
			//set alpha to a neutral value if lowest nodes in
			//recursion-tree are visited without a result
			alpha = 0;
		}else if(recurDepth<MAX_RECURDEPTH && !winnerFound){
			for(int i=0; i<field.WIDTH; i++){
				if(field.columnHeight[i]<field.HEIGHT){
					int tmpVal = minAlphaBeta(3-player, i, alpha, beta);
					if(alpha==(-MAX_RECURDEPTH-1)){
						alpha = tmpVal;
					}else if(alpha>tmpVal){
						alpha = tmpVal;
					}

					if(alpha>=beta){
						break;
					}

				}
			}
		}

		recurDepth--;
		field.removeCoin(column);
		return alpha;
	}

	private int minAlphaBeta(int player, int column, int alpha, int beta){
		boolean winnerFound = false;

		if(field.columnHeight[column]>=field.HEIGHT)
			return 0;

		recurDepth++;
		field.placeCoin(column, player);

		if(checkFour()>0){
			//min-player has 4 in a row
			beta = -MAX_RECURDEPTH-1+recurDepth;
			winnerFound = true;
		}

		if(recurDepth==MAX_RECURDEPTH && !winnerFound){
			//set beta to a neutral value if lowest nodes in
			//recursion-tree are visited without a result
			beta = 0;
		}else if(recurDepth<MAX_RECURDEPTH && !winnerFound){
				for(int i=0; i<field.WIDTH; i++){
					if(field.columnHeight[i]<field.HEIGHT){
						int tmpVal = maxAlphaBeta(3-player, i, alpha, beta);
						if(beta==(MAX_RECURDEPTH+1)){
							beta = tmpVal;
						}else if(beta<tmpVal){
							beta = tmpVal;
						}

						if(beta<=alpha){
							break;
						}
```

```
                                        }
                                }
                        }

                        recurDepth--;
                        field.removeCoin(column);

                        return beta;

                }


                /**
                 *
                 * @param player
                 * @param column
                 * @return
                 */
                private int max(int player, int column){
                        int value = 0;
                        if(field.columnHeight[column]>=field.HEIGHT)
                                return 0;

                        recurDepth++;
                        field.placeCoin(column, player);

                        if(checkFour()>0){
                                value = MAX_RECURDEPTH+1-recurDepth;
                        }

                        if(recurDepth<MAX_RECURDEPTH && value==0){
                                value = -MAX_RECURDEPTH-1;
                                for(int x=0; x<field.WIDTH; x++){
                                        if(field.columnHeight[x]<field.HEIGHT){
                                                int tmpVal = min(3-player,x);
                                                if(value==(-MAX_RECURDEPTH-1)){
                                                        value = tmpVal;
                                                }else if(tmpVal<value){
                                                        value = tmpVal;
                                                }
                                        }
                                }
                        }

                        recurDepth--;
                        field.removeCoin(column);

                        return value;
                }

                /**
                 *
                 * @param player
                 * @param column
                 * @return
                 */
                private int min(int player, int column){
                        int value = 0;
                        if(field.columnHeight[column]>=field.HEIGHT)
                                return 0;

                        recurDepth++;
                        field.placeCoin(column, player);

                        if(checkFour()>0){
                                value = -MAX_RECURDEPTH-1+recurDepth;
                        }

                        if(recurDepth<MAX_RECURDEPTH && value==0){
                                value = MAX_RECURDEPTH+1;
                                for(int x=0; x<field.WIDTH; x++){
```

```
                                        if(field.columnHeight[x]<field.HEIGHT){
                                                int tmpVal = max(3-player,x);
                                                if(value==(MAX_RECURDEPTH+1)){
                                                        value =tmpVal;
                                                }else if(tmpVal>value){
                                                        value = tmpVal;
                                                }
                                        }
                                }
                        }
                        recurDepth--;
                        field.removeCoin(column);

                        return value;
        }


        /**
         * Checks if one of the two player has won, meaning
         * if one of the two has 4 coins in one of the directions.
         * @return the player which has 4 in a row (1 or 2)
         */
        private int checkFour(){
                int num, player;

                //horizontal check
                for(int y=0; y<field.HEIGHT; y++){
                        num=0; player=0;
                        for(int x=0; x<field.WIDTH; x++){
                                if(field.getPlayer(x,y)==player){
                                        num++;
                                }else{
                                        num=1;
                                        player = field.getPlayer(x,y);
                                }
                                if(num==4 && player>0){
                                        return player;
                                }
                        }
                }

                //vertical check
                for(int x=0; x<field.WIDTH; x++){
                        num=0; player=0;
                        for(int y=0; y<field.HEIGHT; y++){
                                if(field.getPlayer(x,y)==player){
                                        num++;
                                }else{
                                        num=1;
                                        player = field.getPlayer(x,y);
                                }
                                if(num==4 && player>0){
                                        return player;
                                }
                        }
                }

                //diagonally up
                /*
                 *  1  2  3  4  5  6  7
* [ ][ ][ ][#][#][ ][ ]
* [ ][ ][#][#][ ][ ][ ]
* [ ][#][#][ ][ ][ ][ ]
* [#][#][ ][ ][ ][ ][ ]
* [#][ ][ ][ ][ ][ ][ ]
* [ ][ ][ ][ ][ ][ ][ ]
*  1  2  3  4  5  6  7
                 *
                 */
             for(int xStart=0, yStart=2; xStart<4;){
                        num=0; player=0;
```

```
                for(int x=xStart, y=yStart; x<field.WIDTH && y<field.HEIGHT; x++, y++){
                        if(field.getPlayer(x,y)==player){
                                num++;
                        }else{
                                num=1;
                                player = field.getPlayer(x,y);
                        }
                        if(num==4 && player>0)
                                return player;
                }
                if(yStart==0)
                        xStart++;
                else
                        yStart--;
            }


                //diagonally down
                /*
                 *  1  2  3  4  5  6  7
          * [ ][ ][ ][ ][ ][ ][ ]
          * [#][ ][ ][ ][ ][ ][ ]
          * [#][#][ ][ ][ ][ ][ ]
          * [ ][#][#][ ][ ][ ][ ]
          * [ ][ ][#][#][ ][ ][ ]
          * [ ][ ][ ][#][#][ ][ ]
          *  1  2  3  4  5  6  7
                 *
                 */
            for(int xStart=0, yStart=3; xStart<4;){
                num=0; player=0;
                for(int x=xStart, y=yStart; x<field.WIDTH && y>=0; x++, y--){
                        if(field.getPlayer(x,y)==player){
                                num++;
                        }else{
                                num=1;
                                player = field.getPlayer(x,y);
                        }
                        if(num==4 && player>0)
                                return player;
                }
                if(yStart==field.HEIGHT-1)
                        xStart++;
                else
                        yStart++;
            }
            return 0;
          }

          /**
           * This method reads the input from the user, in which column he wants to put
           * his coin.
           * @param player
           * @return
           */
          private int getColumnInput(int player) {
                if(player==1)
                        System.out.print("It's your turn " + opponents[player-1] + "(your sign=O):\nChoose
a column where to put the coin (1-" + field.WIDTH + "): ");
                else
                        System.out.print("It's your turn " + opponents[player-1] + "(your sign=X):\nChoose
a column where to put the coin (1-" + field.WIDTH + "): ");

                BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
                try {
                        String tmp=in.readLine();
                        return Integer.parseInt(tmp);
                } catch(Exception ex){
                        return -1;
                }
           }
```

```
/**
 * helper method for readPlayernames(..)
 * It basically reads a string from the command-line input
 * @return
 */
private String readString(){
          BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
          try {
                    String tmp=in.readLine();
                    return tmp;
          } catch(Exception ex){
   return null;
          }
}

/**
 * Reads the playername of either 1 or both players, depending
 * whether the boolean "againCPU" is true or false
 * @param againCPU
 */
private void readPlayernames(boolean againCPU){
          if(!againCPU){
                    String player1 = null;
                    String player2 = null;
                    do{
                              System.out.print("Please input the name of the 1st player: ");
                              player1 = readString();
                    }while(player1==null);

                    do{
                              System.out.print("Please input the name of the 2nd player: ");
                              player2 = readString();
                    }while(player2==null);

                    opponents[0] = player1;
                    opponents[1] = player2;
          }else{
                    String playername = null;
                    do{
                              System.out.print("Please input your name: ");
                              playername = readString();
                    }while(playername==null);
                    opponents[0] = playername;
                    opponents[1] = "CPU";
          }
          System.out.println("\n");
}


/**
 * reads the level of the CPU --> the depth of the recursion in the
 * min-max algorithm from the keyboard (from the user)
 * @return
 */
private int readCPUSkills(){
          int skill = 0;
          BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
          do{
                    System.out.print("Please input level of CPU skill (1=low, 8=high): ");
                    try {
                              String tmp=in.readLine();
                              skill =  Integer.parseInt(tmp);
                    } catch(Exception ex){
                              skill = 0;
                    }
          }while(skill<1 || skill>8);
          return skill;
}

/**
```

```
	 * Estimates which of the two players starts
	 * @return the number of the player which starts (1 or 2)
	 */
	private int randomPlayer(){
		Random r = new Random();
		int playerToStart = -1;
		do{
			playerToStart = r.nextInt(3);
		}while(playerToStart<1 || playerToStart>2);
		return playerToStart;
	}


}
```

GameField.java

```
**
 * This class represents the whole game board. It is
 * a two-dimensional array containing "Cell" objects.
 *
 * @author Juri Strumpfohner
 *
 */
public class GameField {
	public final int HEIGHT = 6;
	public final int WIDTH = 7;
	public int[] columnHeight;
	private Cell[][] grid = new Cell[HEIGHT][WIDTH];

	public GameField(){
		//initialize the field
		for(int i=0; i<HEIGHT; i++){
			for(int j=0; j<WIDTH; j++){
				grid[i][j] = new Cell();
			}
		}
		columnHeight = new int[WIDTH];
	}

	/**
	 *
	 * @param column = the column where to put the coin
	 * @param player = the player that put the coin
	 */
	public void placeCoin(int column, int player){
		grid[columnHeight[column]][column].placeCoin(player);
		columnHeight[column]++;
	}

	/**
	 * Used in recursion to remove coin after inserting it
	 * for testing.
	 * @param column
	 */
	public void removeCoin(int column){
		columnHeight[column]--;
		grid[columnHeight[column]][column].removeCoin();
	}

	/**
	 * Gets the player of the cell indicated by the column(x) and row(y)
	 * @param column
	 * @param row
	 * @return
	 */
	public int getPlayer(int column, int row){
		return grid[row][column].getPlayer();
	}

	/**
	 * Overwritten toString() to print the game-board to
	 * the console
	 */
	public String toString(){
		StringBuffer buffer = new StringBuffer();
		buffer = buffer.append("           ");

		//write column headers
		for(int i=0; i<WIDTH; i++){
			buffer = buffer.append(i+1 + "  ");
		}
		buffer = buffer.append("\n");

		//writing grid
```

```
                    for(int i=HEIGHT-1; i>=0; i--){
                              buffer = buffer.append("          ");
                              for(int j=0; j<WIDTH; j++){
                                        buffer = buffer.append(grid[i][j].toString());
                              }
                              buffer = buffer.append("\n");
                    }

                    //write column headers
                    buffer = buffer.append("           ");
                    for(int i=0; i<WIDTH; i++){
                              buffer = buffer.append(i+1 + "  ");
                    }
                    return buffer.toString();
          }
}
```

Cell.java

```
/**
 * This class represents a cell in the GameBoard
 *
 * @author Juri Strumpflohner
 *
 */
public class Cell {
          private boolean isOccupied;
          private int player; //1 --> 1st player, 2 --> 2nd player(CPU)

          public Cell(){
                    isOccupied = false;
                    player = -1;
          }

          public void placeCoin(int playerNum){
                    isOccupied = true;
                    player = playerNum;
          }

          public void removeCoin(){
                    player = -1;
                    isOccupied = false;
          }

          public int getPlayer(){
                    return player;
          }

          public String toString(){
                    String tmp;
                    if(!isOccupied){
                              tmp = "[ ]";
                    }else{
                              if(player==1)
                                        tmp = "[O]";
                              else
                                        tmp = "[X]";
                    }
                    return tmp;
          }
}
```