## Exercise 12.4

1. A 2D array is an array with both rows and columns.

*Example: List[rows][columns]*

1. **int** [][] table;

table = **new** **int**[10][20];

1. **int** [][] table;

**int** row =0;

**int** col = 0;

table = **new** **int**[10][10];

**for** (**int** i=0; i < table.length; i++)

**for** ( **int** j = 0; j< table[i].length; j++)

**if** (table[i][j] <0)

{

row = i;

col = j;

System.***out***.println("Negitve Number is at list[" + row +"]["+col+"]");

}

**if** (row <= 0 && col <= 0){

row= col =table.length;

System.***out***.println( "No Negitive!");

}

1. times each row by each col
2. **int** [][] table;

table = **new** **int**[10][20];

**for** (**int** i=0; i < table.length; i++){

**for** ( **int** j = 0; j< table[i].length; j++) System.***out***.print(table[i][j] + " ");

System.***out***.println();

}

//chapter 12-5

**boolean** success = board.placeXorO(player, row, column);

**if** (!success)

System.***out***.println("Error: cell already occupied!");

**else**{

System.***out***.println(board);

**if** (player == 'X')

player = 'O';

**else**

player = 'X';

**if** (!board.full()){

**int** randomRow, randomcol;

**do**{

randomRow = gen.nextInt(3) +1;

randomcol=gen.nextInt(3) +1;;

success = board.placeXorO(player, randomRow, randomcol);

}**while** (success == **false**);

System.***out***.println(board);

}

## Chapter 12-13

**import** java.util.\*;

**public** **class** PennyPitch {

**private** Object[][] board; // The tic-tac-toe board

// Instantiate an empty board

**public** PennyPitch(){

board = **new** Object[(**int**)Math.*random*()][(**int**)Math.*random*()];

reset();

}

// Reset the board to empty

**public** **void** reset(){

**for** (**int** row = 0; row < 3; row++)

**for** (**int** column = 0; column < 3; column++)

board[row][column] = 1;

}

// Return a string representation of the board

**public** String toString(){

String result = "\n";

**for** (**int** row = 0; row < 3; row++){

**for** (**int** column = 0; column < 3; column++)

result += board[row][column] + " ";

result += "\n";

}

**return** result;

}

// If the move is legal, make it and return true, else return false

**public** **boolean** placeXorO(**char** s, **int** row, **int** column){

**if** (board[row - 1][column - 1] == '-'){

board[row - 1][column - 1] = s;

**return** **true**;

}**else**

**return** **false**;

}

// Return true if the board is full else return false

**public** **boolean** full(){

**for** (**int** row = 0; row < 3; row++)

**for** (**int** column = 0; column < 3; column++)

**if** (board[row][column] == '-')

**return** **false**;

**return** **true**;

}

// Return '-' if there is no winner, else return 'X' or 'O'

**public** **char** getWinner(){

// Each triple represents the current content of a row,

// column, or diagonal

String[] triples = **new** String[8];

// Obtain the current content of the three rows

**for** (**int** column = 0; column < 3; column ++)

triples[column] = getColumn(column);

// Add in the content of the three columns

**for** (**int** row = 0; row < 3; row ++)

triples[3 + row] = getRow(row);

// Add in the two diagonals

triples[6] = getDiagonal(0);

triples[7] = getDiagonal(2);

// If any triple equal "**XXX**", then

// return 'X'

// Else if any triple equals "OOO" then

// return 'O'

// Else

// return '-"

}

// Return the content of a column

**private** String getColumn(**int** column){

String result = "";

**for** (**int** row = 0; row < 3; row++)

result += board[row][column];

**return** result;

}

// Return the content of a row

**private** String getRow(**int** row){

String result = "";

**for** (**int** column = 0; column < 3; column++)

result += board[row][column];

**return** result;

}

// Return the content of a diagonal

**private** String getDiagonal(**int** startRow){

String result = "";

**if** (startRow == 0)

**for** (**int** i = 0; i < 3; i++)

result += board[i][i];

**else**

**for** (**int** i = 0; i < 3; i++)

result += board[2 - i][i];

**return** result;

}

}

## Review

1. binary search
2. bubble sort
3. A.
4. Logical size
5. A.