Charlie Comeau and Nikhil Havanur

Mancala.java

May 22, 2017

Hatcher

**Executive Summary // Charlie**

1. **Overview**

* Creating Mancala in java
* One person game against artificial intelligence computer (Math.random())

1. **Problem Statement**

* Using our collective knowledge of the Java language, how do we accurately and efficiently model the game “Mancala” on java?

1. **Rules**

* Pebble movement procedure
* Additional pebble captures
* Instructions appear in both English and Spanish

1. **Creation Process**

* Need for Scanner class
* 2 separate files: 1 class file and 1 runner

1. **Class File Components**

* The role of each private instance variable
  + Int[][] game
  + Int yourBank
  + Int oppBank
  + Boolean goAgain

1. **Format**

* Outfitting the game board as a 2-D array

1. **Methods**

* The role of each method in the class file
* Each method follows the rules outlined
  + Mancala Constructor
  + yourMove(int row, int col)
  + oppMove()
  + getYourBank()
  + getOppBank()
  + getGoAgain()
  + checkEnd()
  + finalCollect()
  + findWinner()
  + toString()

1. **Runner File Components**

* Interactions of all the methods and loops to create actual game

1. **Troubleshooting Process**

* Compare code output with actual Mancala board

1. **Conclusion**

* Most of the work should be done in the design phase

Be diligent and strategic in your code testing

**Generic Overview**

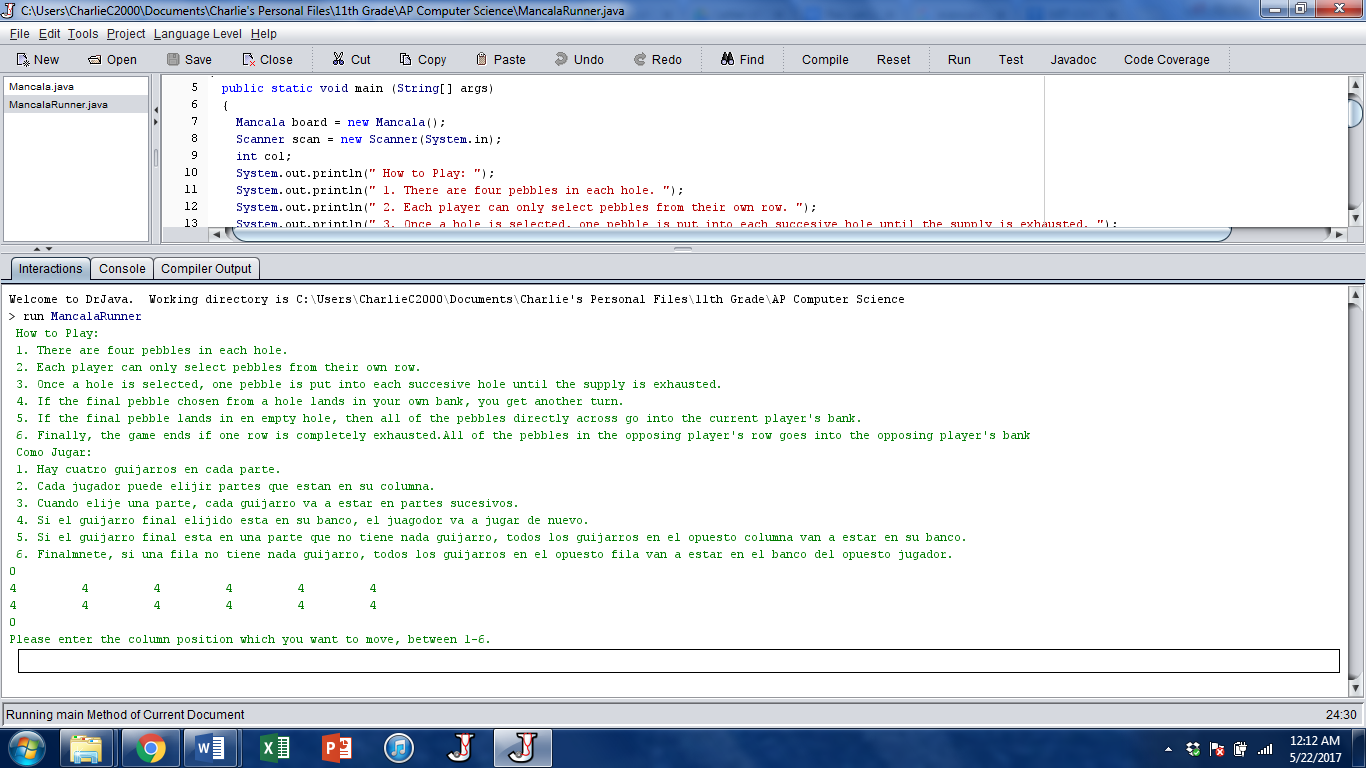
In our end-of-year project, our teacher, Mr. Hatcher, gave us the flexibility to create almost anything related to the field of computer science, ranging from websites, GUI’s, games, to robotics, which is to be determined by our own mutual interests. After a thorough evaluation of the choices at hand, we settled upon trying to model the game “Mancala” on java. As a result, we had to create our methods in a way that respects and follows the various rules of the game. This situation leads us to our problem statement: Using our collective knowledge of the Java language, how do we accurately and efficiently model the game “Mancala” on java?

**Rules of Mancala.java //Nikhil**

1. Each hole contains four pebbles and zero pebbles in both of the banks at the start of the game.
2. Players can only select pebbles from their own row (labeled 1-6).
3. Once a hole is selected the pebbles will be dropped into each successive hole until the resource is exhausted.
4. If the final pebble lands in the player’s bank, then the player receives another turn.
5. If the final pebble lands in an empty hole, then the all of the pebbles directly across from the empty hole go into the player’s bank.
6. The game ends when all of the pebbles have been exhausted from one row. When this occurs, all of the remaining pebbles in the opposite row go to the opposing player’s bank.

-Player with the most marbles at the end WINS.

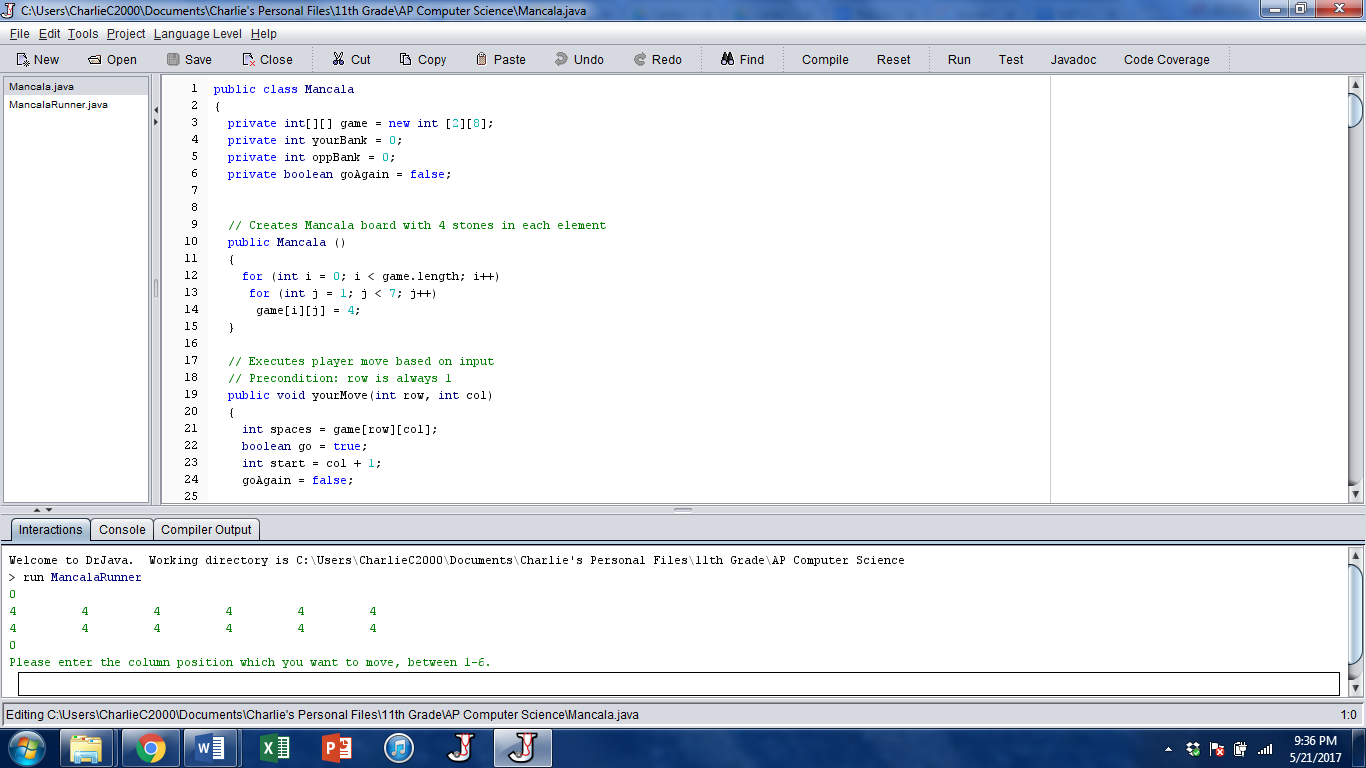
-The rules are also provided in the program Mancala.java both in English and Spanish.

This is the instructional blurb that comes up when the program is started.

**Creation Process //Nikhil**

We decided to have Mancala.java be a one person application, where the user is playing against an artificial intelligence opponent, and the game is played in the interactions panel with manual input from the user. As a result, we realized that it would be easiest to have 2 separate files: a class file containing all the methods, and a runner program where the game is played. This in turn made us ask ourselves what kind of instance variables and methods we would need. From there, we used the rules to determine the specific needs in which to what we needed to create.

**Class File Components // Charlie**

In our class file, we have various different private instance variables that each serve a different purpose. This is a picture of our private instance data in the class file.

**private int[][] game = new int [2][8];**

This is the actual game board of Mancala.java, created using a 2-D array of 2 rows and 8 columns. While the end holes of a real Mancala board cover 2 spaces, the ensuing methods are written to offset this difference.

**private int yourBank = 0;**

This variable represents how many pebbles are in the user’s bank, and initializes it to 0 at the start of the game.

**private int oppBank = 0;**

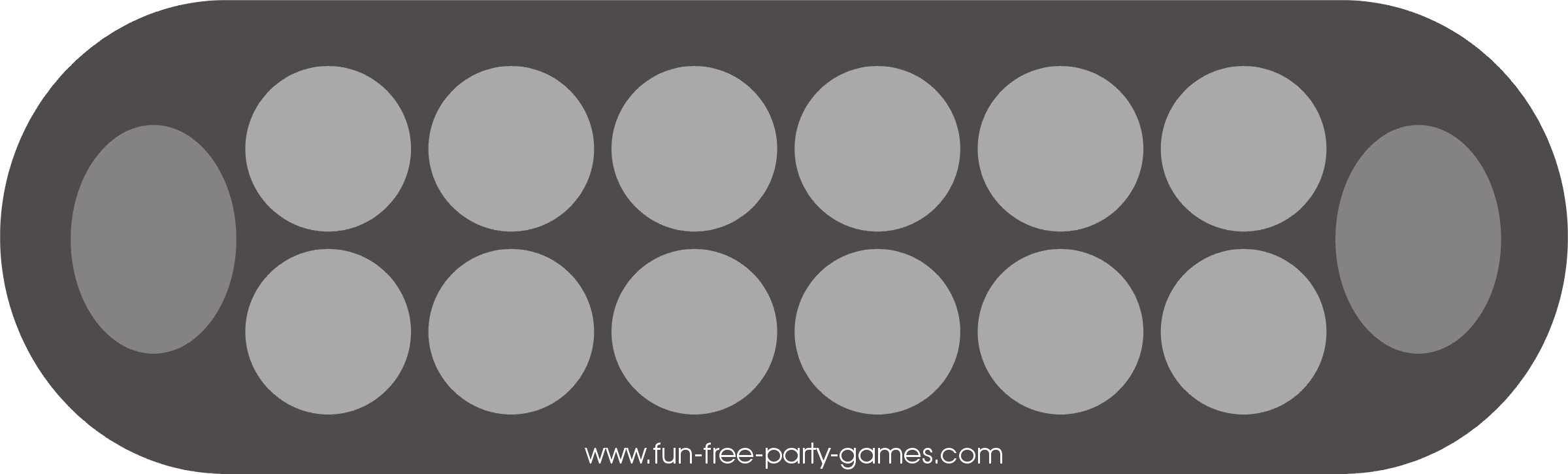
This variable represents how many pebbles are in the computer opponent’s bank, and initializes it to 0 at the start of the game.

**private boolean goAgain = false;**

This variable represents if a player’s move ends directly into his bank, and therefore can or cannot go again.

**Format // Charlie**

As stated before, the game is played in the interactions panel of the runner program. In representing our game board as a 2D array, this is a picture of how we formatted the game board.

**Column: 0 1 2 3 4 5 6 7**

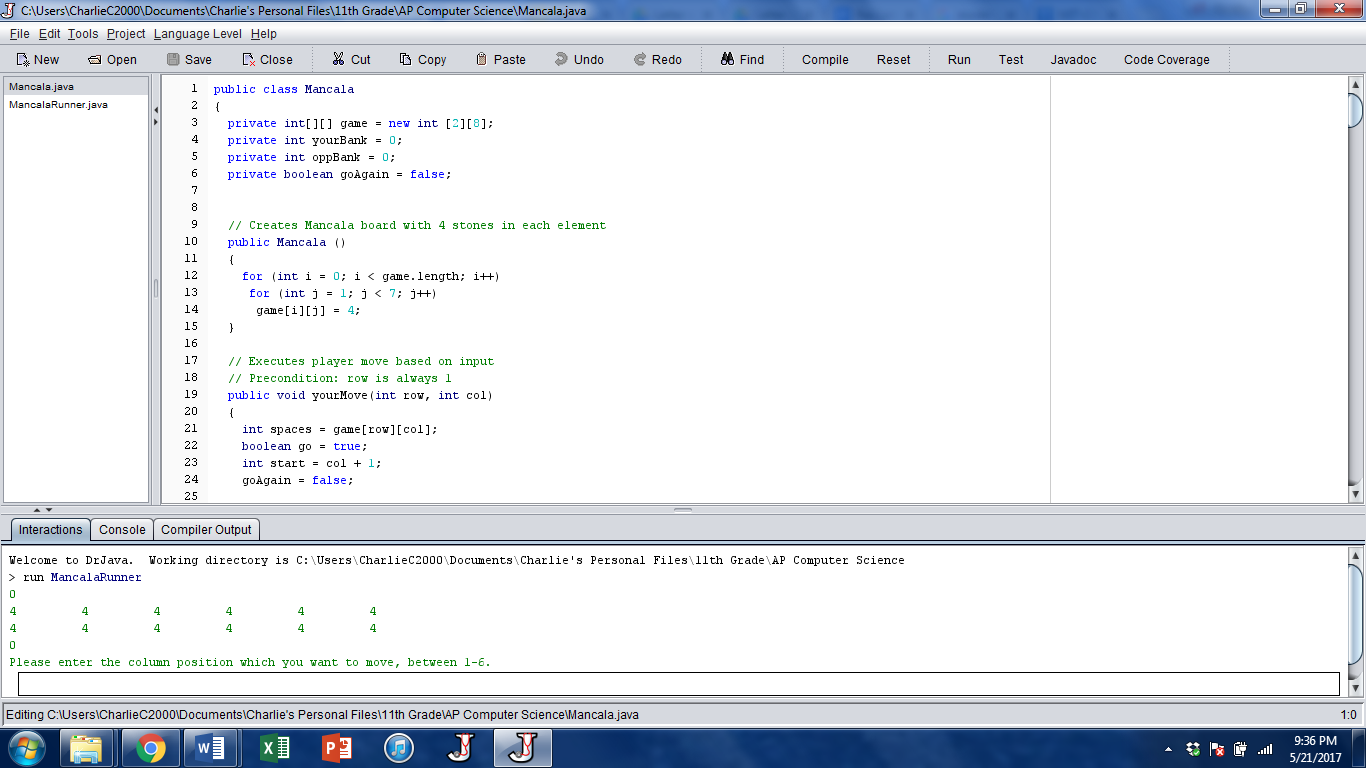
**Row**

**0**

**1**

With this system, each hole can be labeled. Each bank spot covers two holes, but the actual bank values are not in the 2-D array, but instead as the yourBank and oppBank variables. In the code, the bank holes only act as placeholders as the computer traverses through the array. Row 0 is the computer’s row, while row 1 is the user’s row.

This is a picture and diagram of how the board is printed in the interactions panel and the variables printed there.



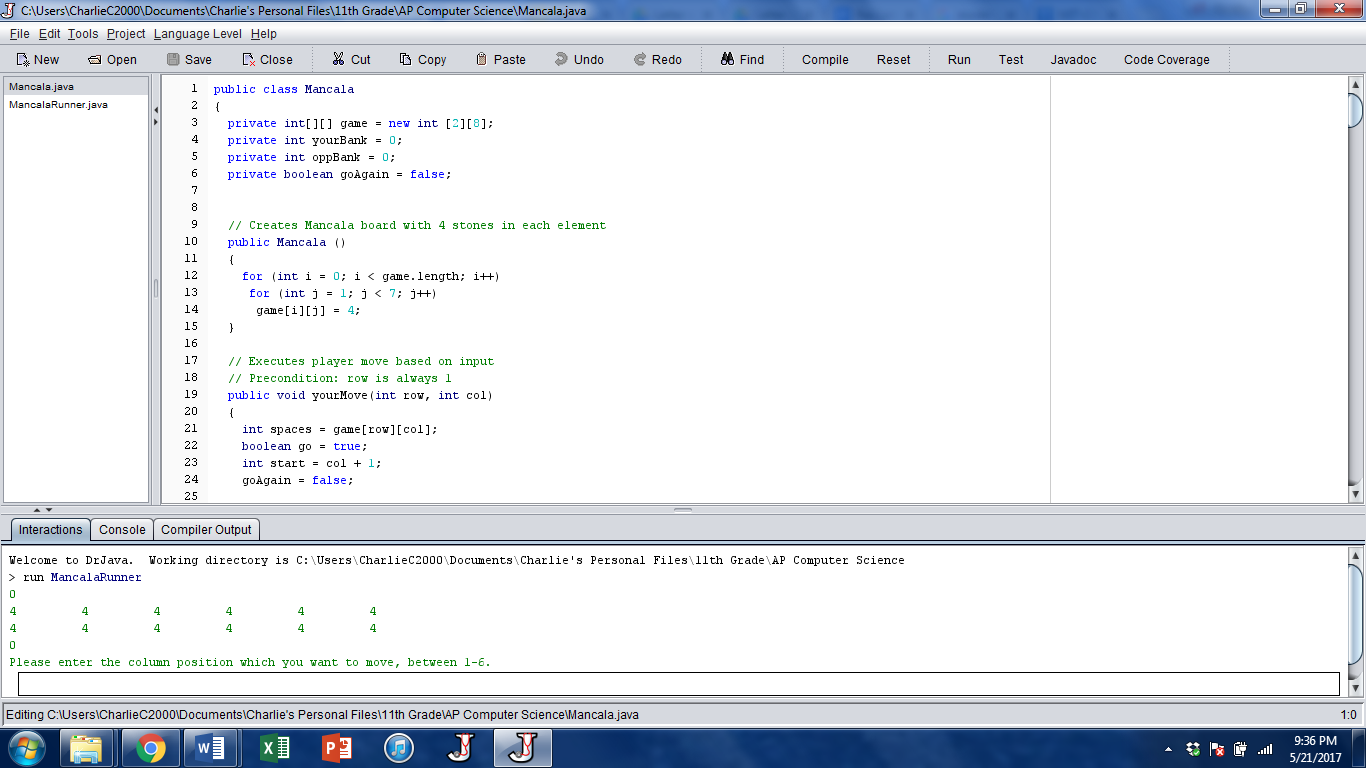
oppBank

yourBank

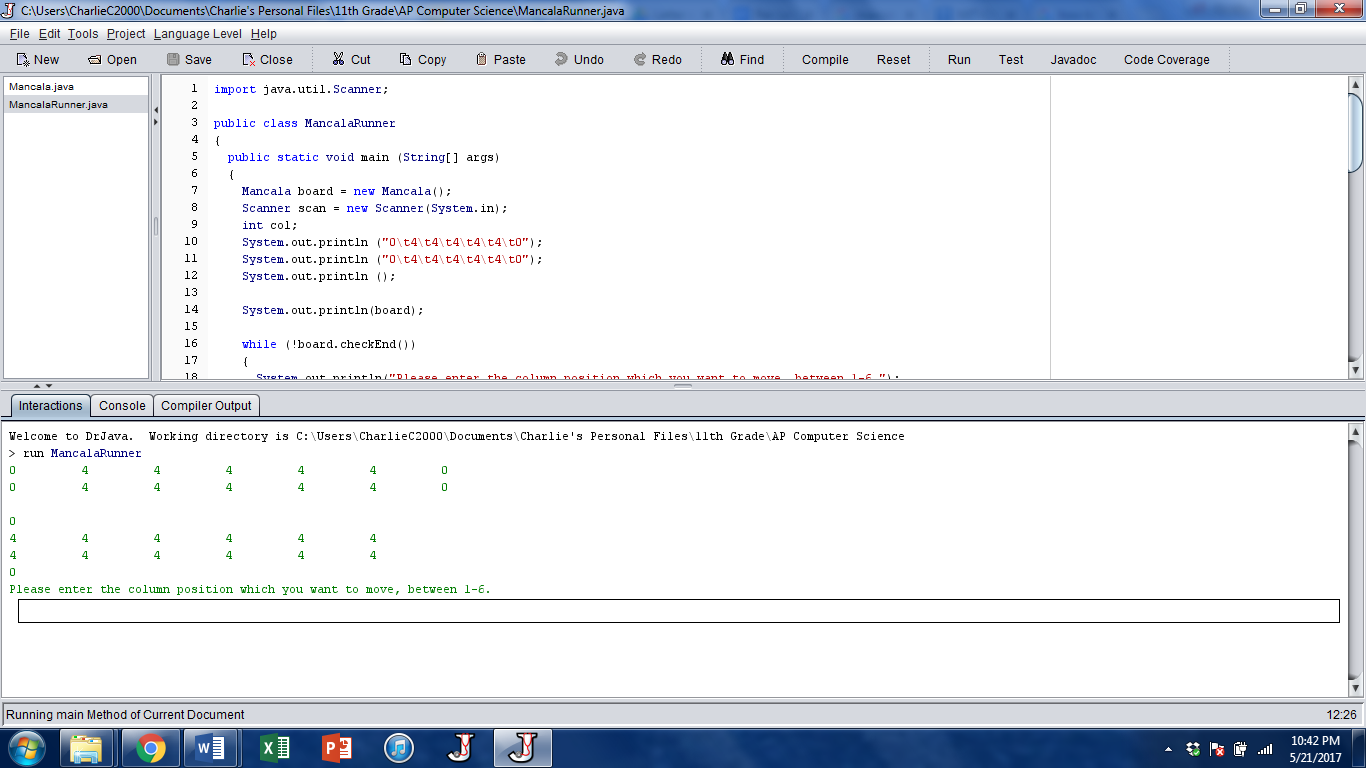
Besides the oppBank and yourBank numbers, each number is representative of how many pebbles are in each hole, with the computer’s row on top and the user’s row on the bottom.

**Methods**

**Mancala Constructor: // Charlie**

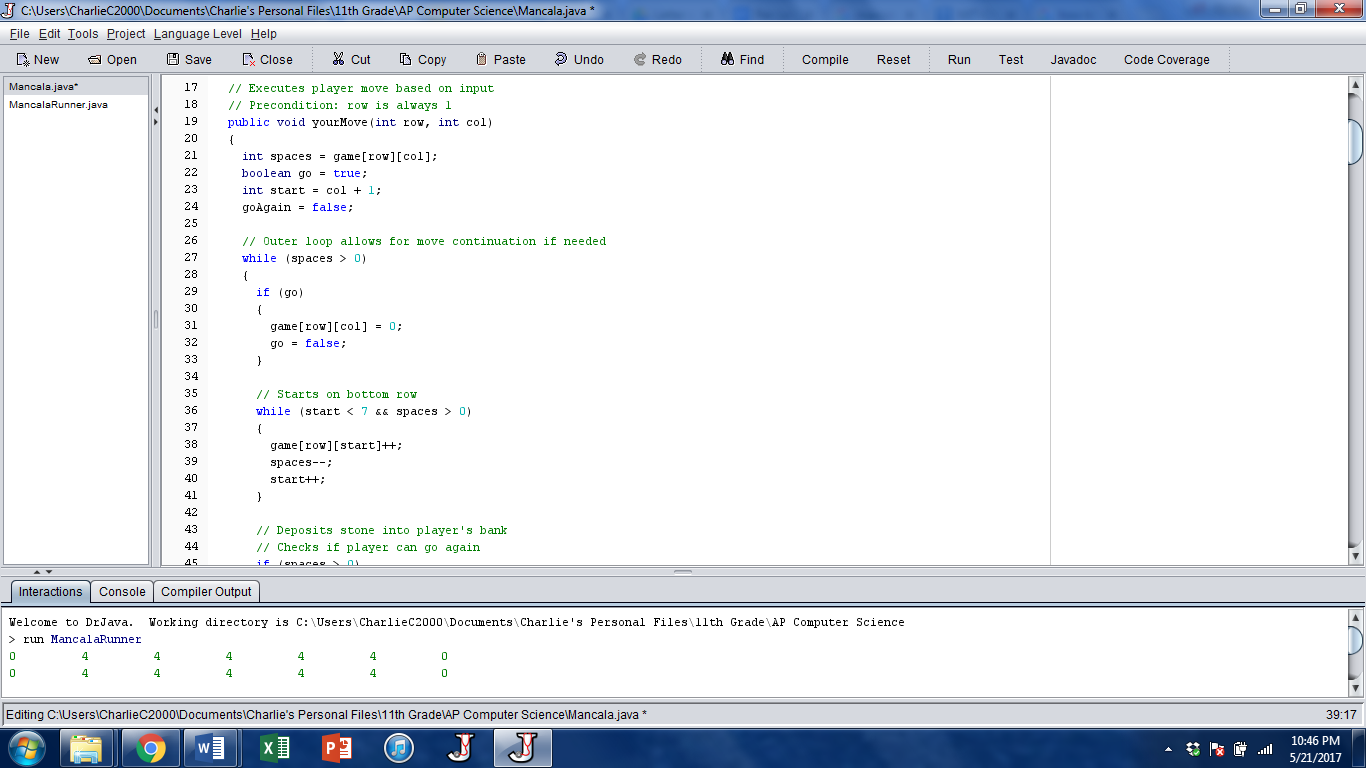


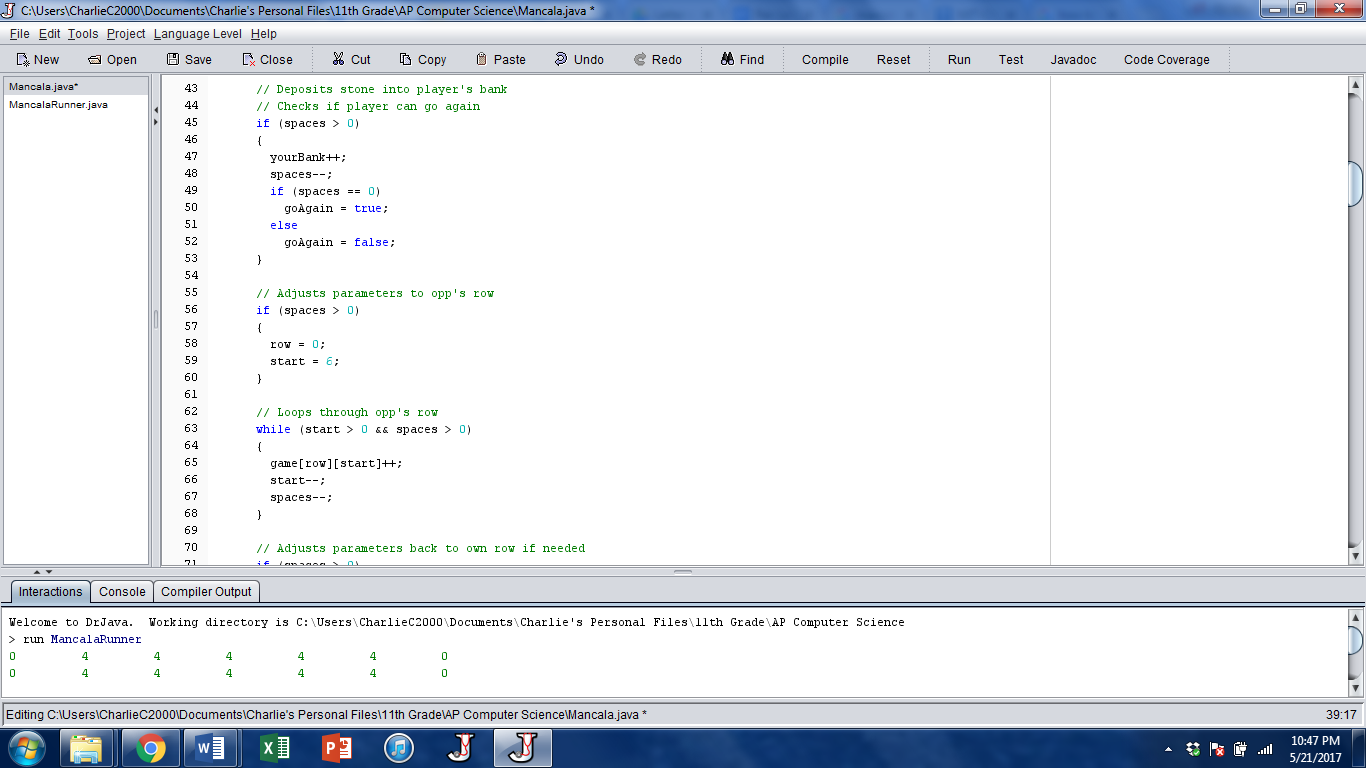
Using a for loop, this sets up the 2D array that acts as the Mancala board. We set the bounds on the nested for loops to populate all the holes except the bank ones at the end, where the column values are either 0 or 7. Once created, the board will look like this.

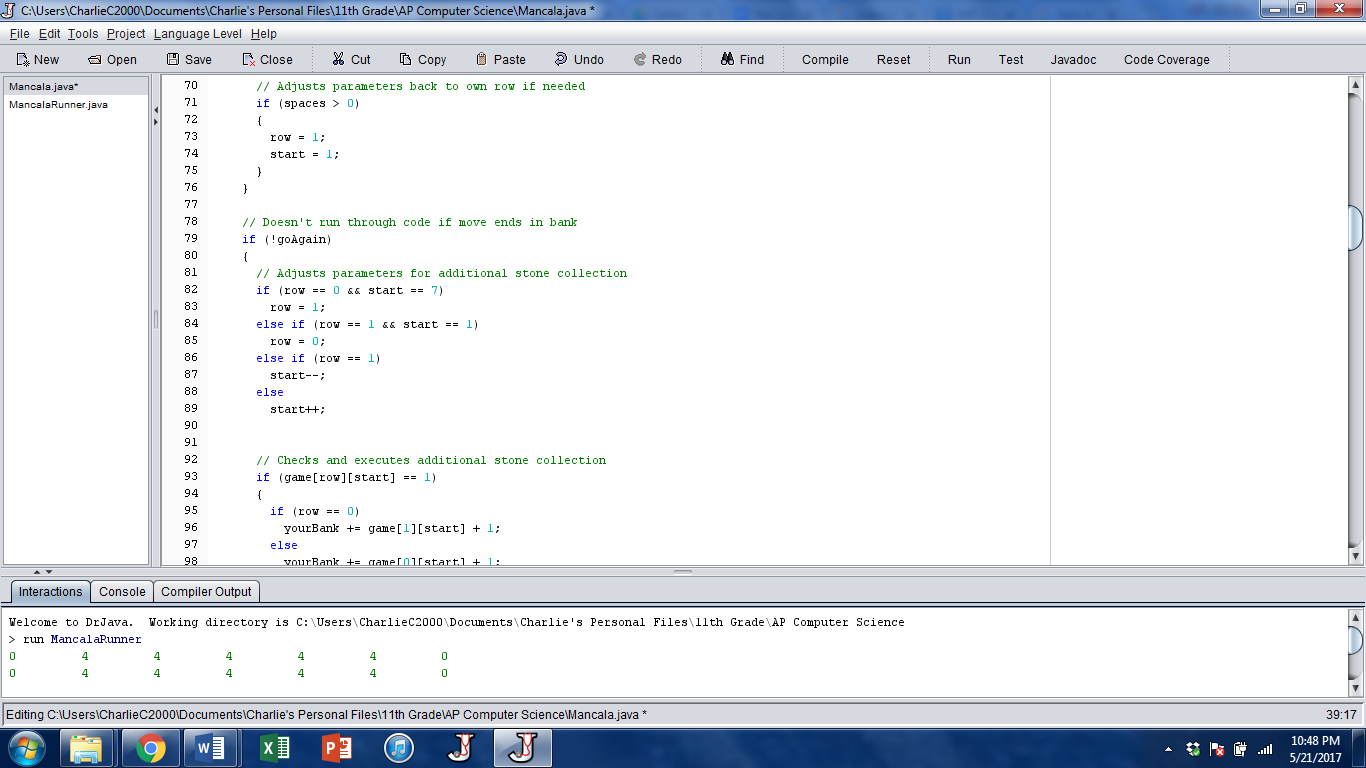


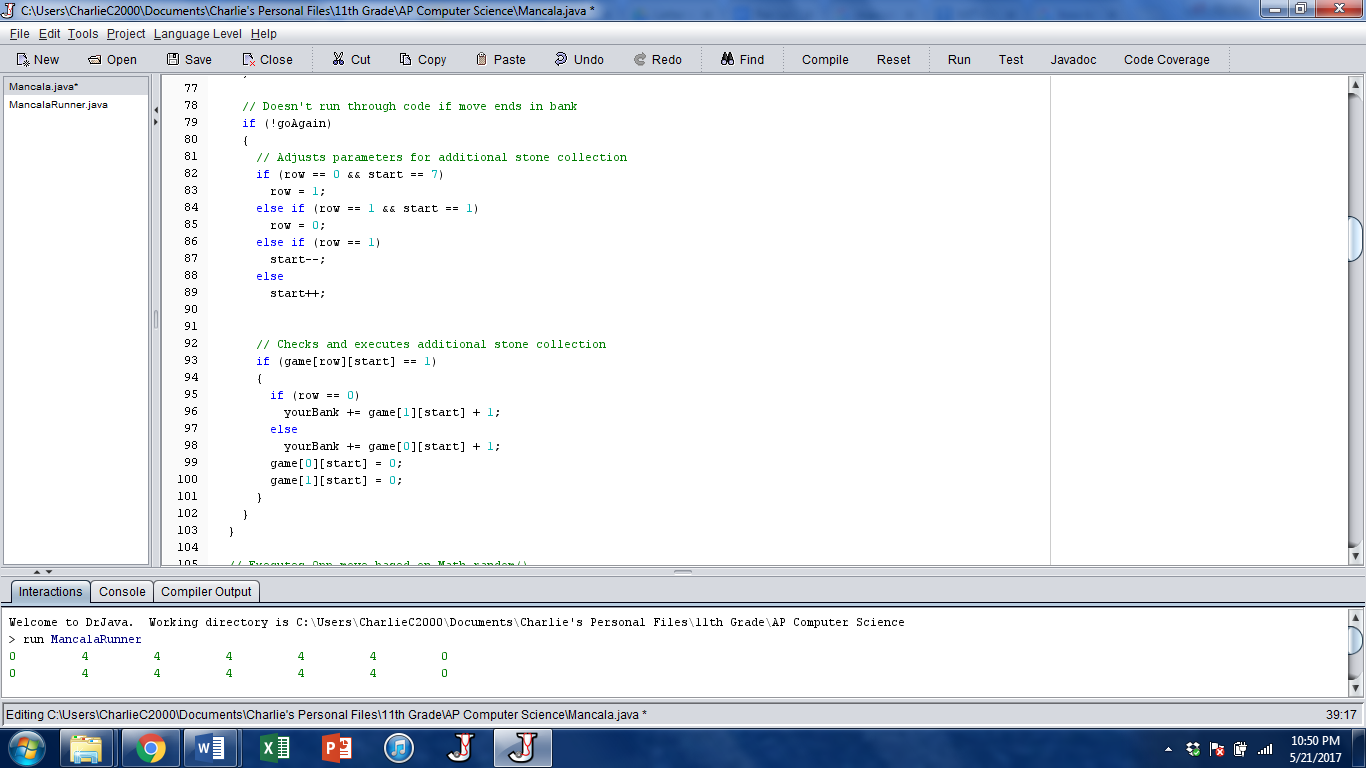
**yourMove(int row, int col) // Charlie**

This enables the user to select which hole the pebbles should originate from, and executes the corresponding move.

Much of the code revolves around the int variable spaces, which represents how many pebbles the user-picked hole contains. The code sets that chosen hole to be 0, as all the pebbles are now gone, and proceeds to deposit a stone in each hole it passes, which decrements the spaces variable, and keeps going until spaces is 0. The code accounts for when the user deposits a stone in their own bank, then traverses through the computer’s row, then skips the computer’s bank, and goes back into the user’s row as needed. The code then checks to see if the user can go again if their move ended in their own bank. Once the move is done, the code checks if the user’s move ended in an empty slot, in which the user captures that pebble and any pebbles directly across from it into their bank. 



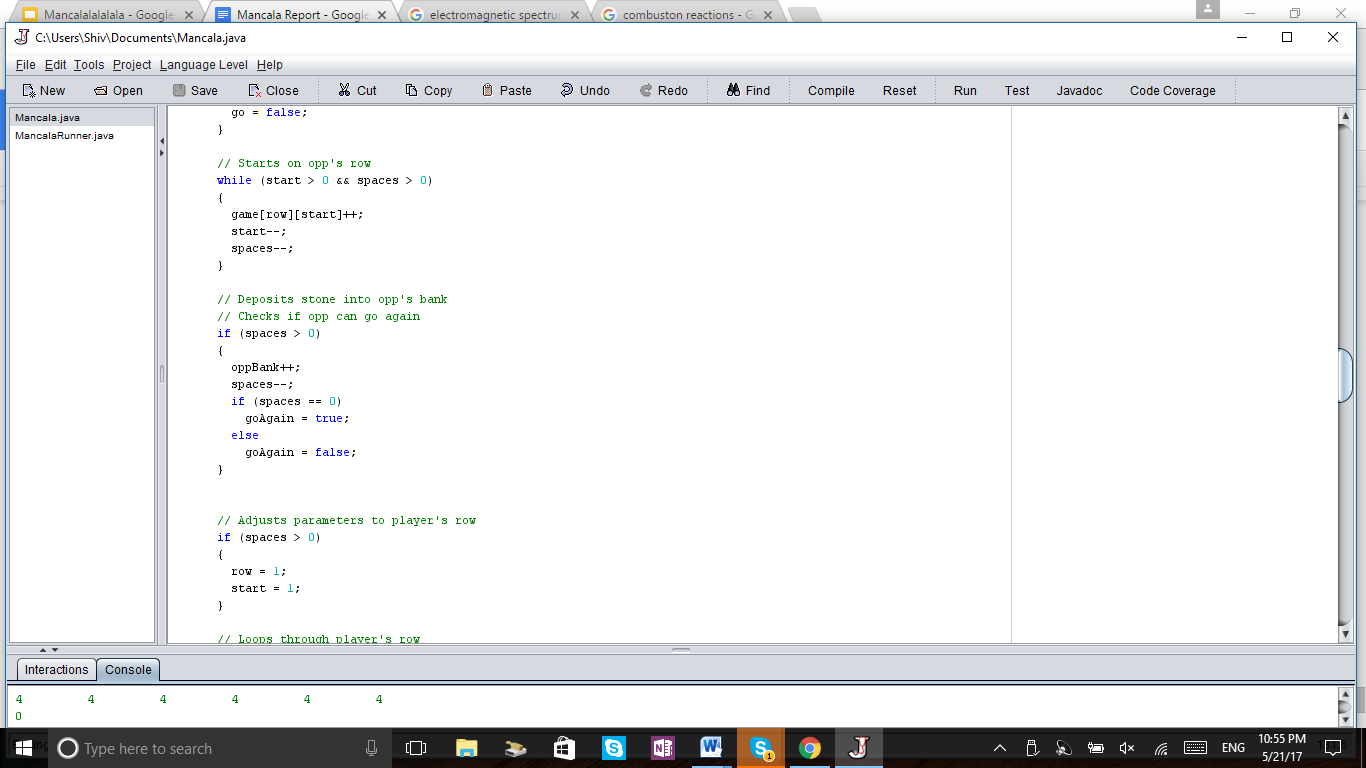
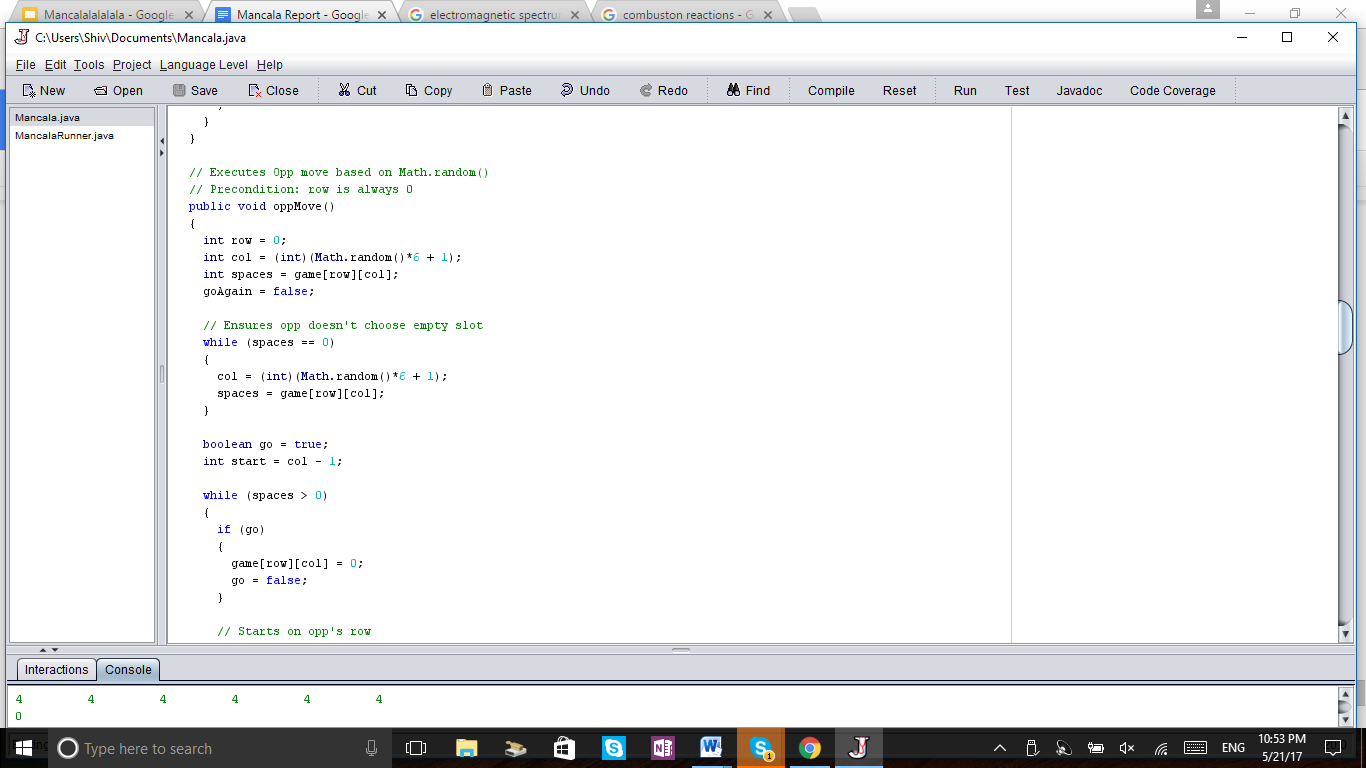


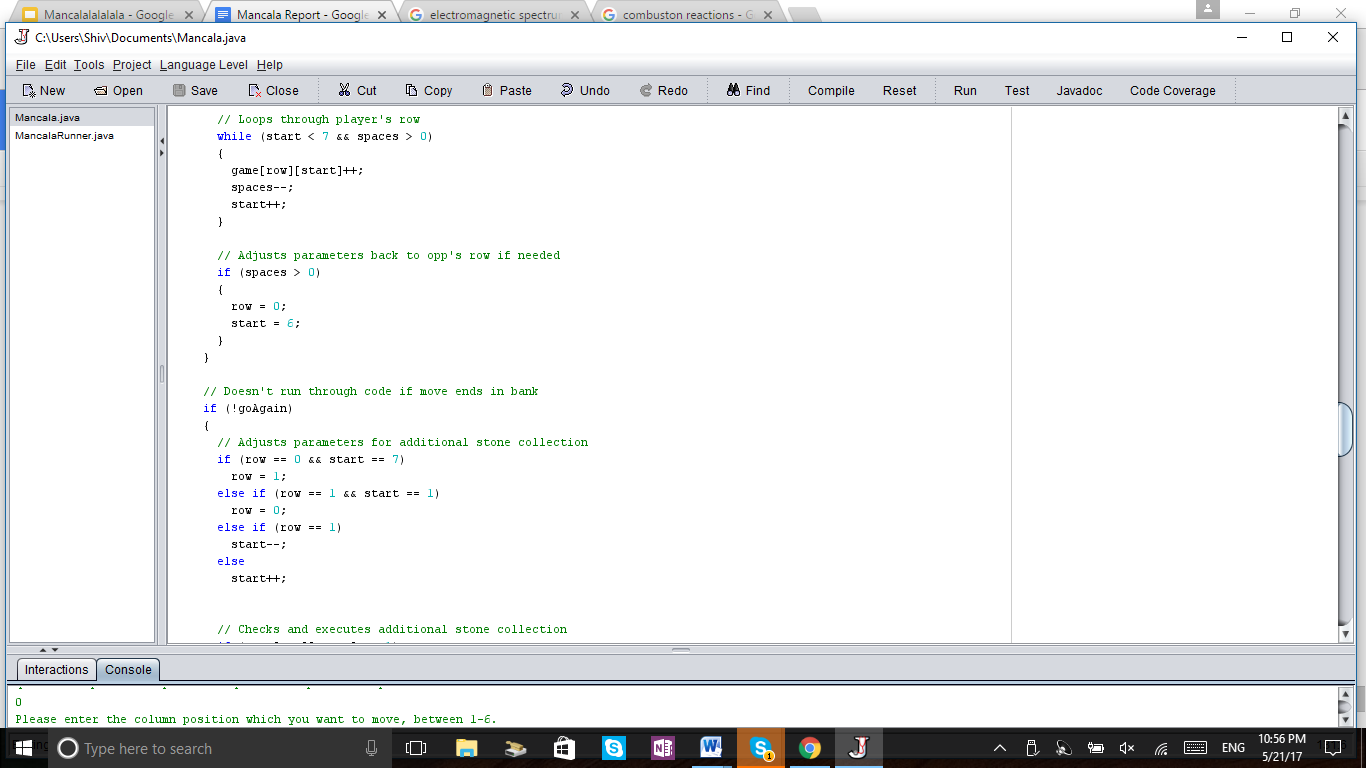


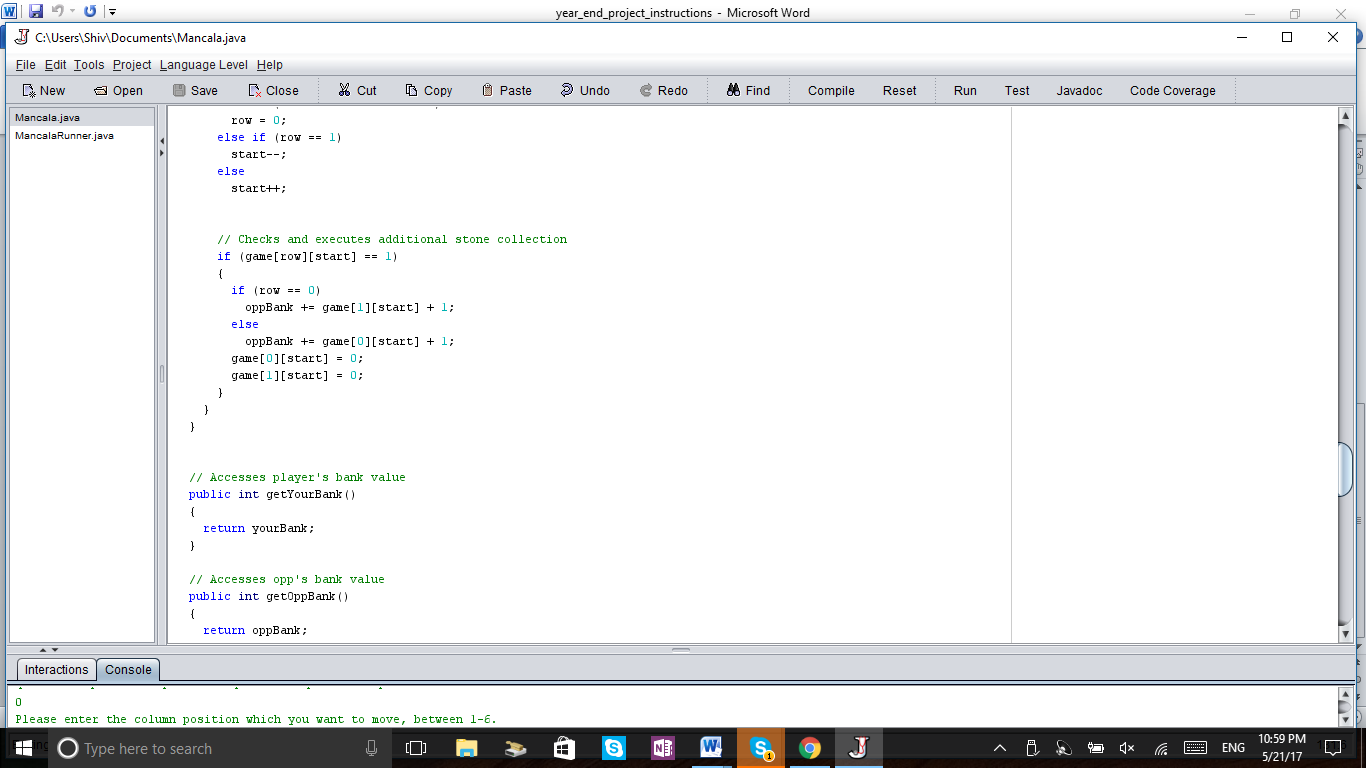
**oppMove() // Charlie**

Contains the Math.random method that allows for the computer to select a hole and executes the corresponding move.

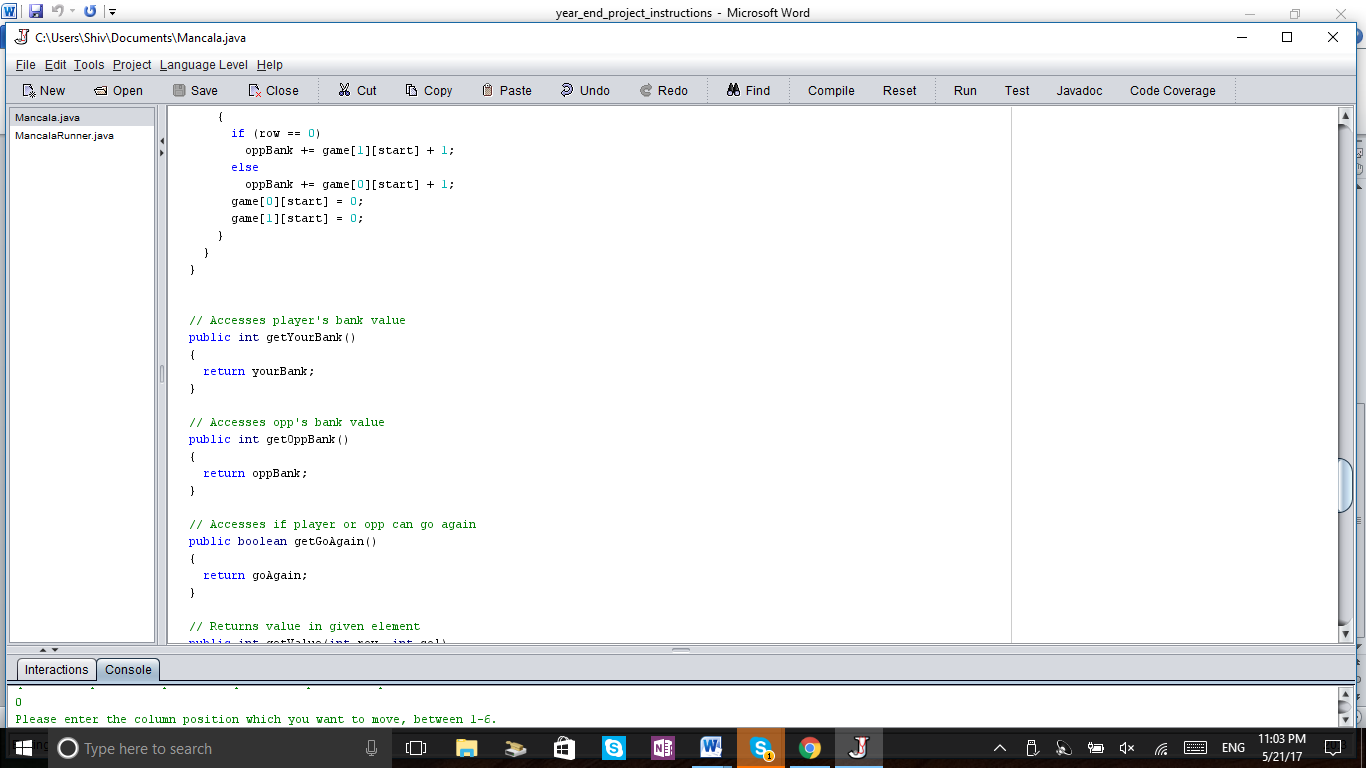
While the user manually inputs their chosen hole, the computer opponent chooses theirs by the Math.random() method, where if the computer chooses a hole with no pebbles, the code keeps looping until the computer chooses one that does have pebbles. From there, the code sets that chosen hole to be 0, as all the pebbles are now gone, and proceeds to deposit a stone in each hole it passes, which decrements the spaces variable, and keeps going until spaces is 0. The code accounts for when the computer deposits a stone in their own bank, then traverses through the user’s row, then skips the user’s bank, and goes back into the computer’s row as needed. The code then checks to see if the computer can go again if their move ended in their own bank. Once the move is done, the code checks if the computer’s move ended in an empty slot, in which the computer captures that pebble and any pebbles directly across from it into their bank.



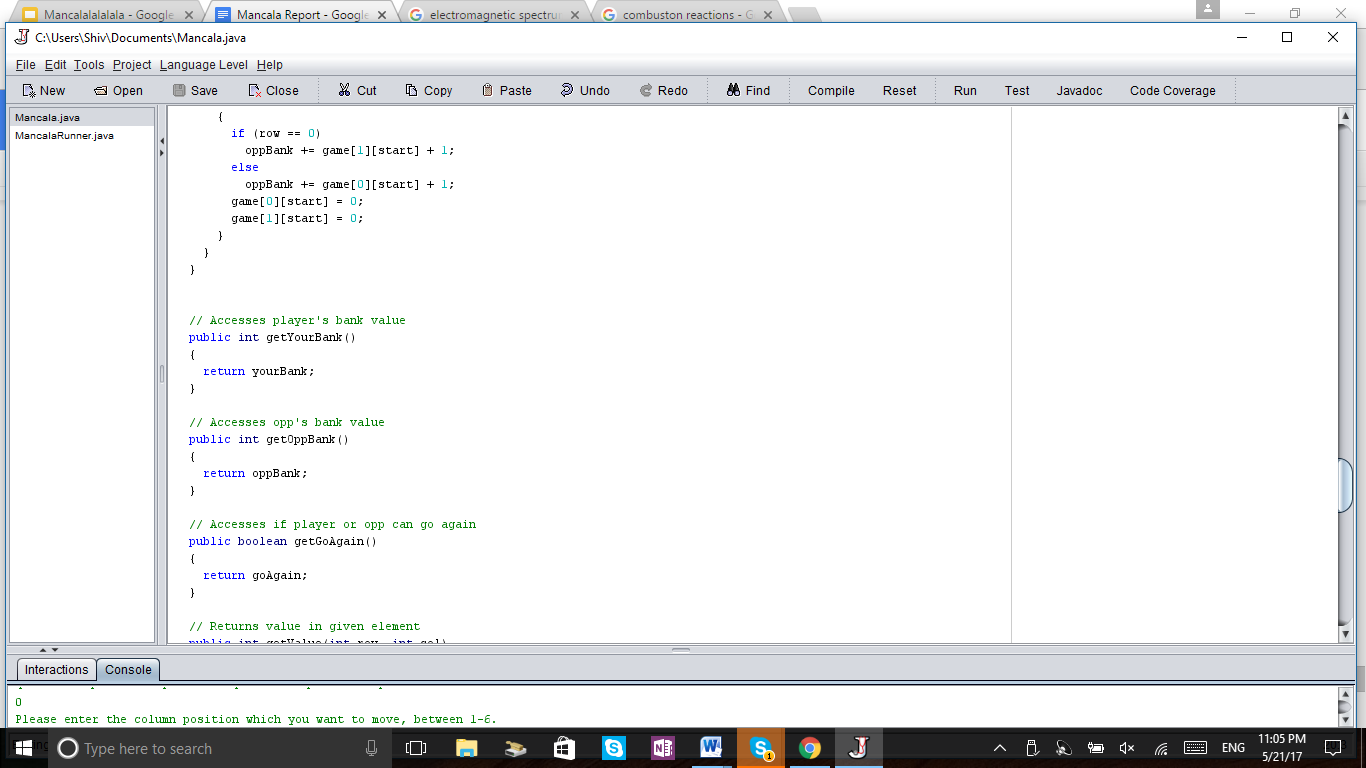




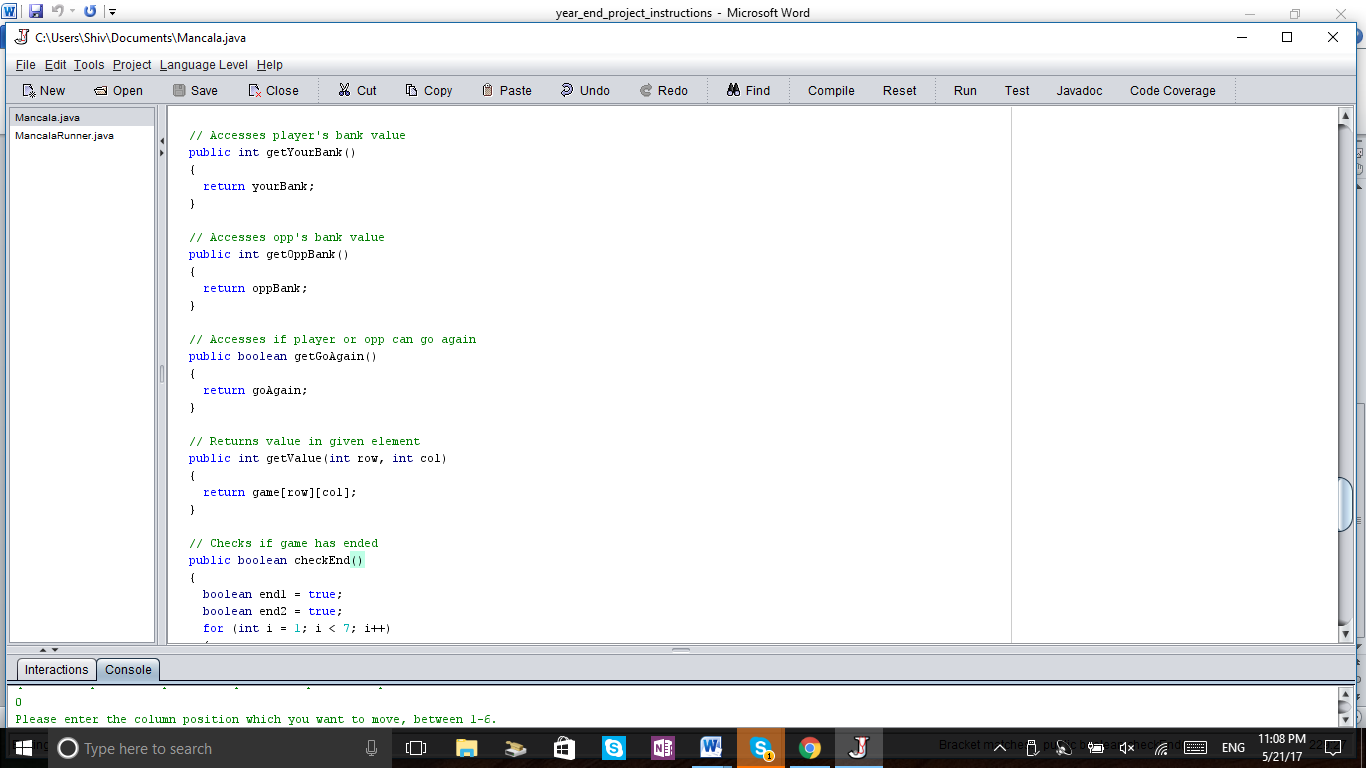
**getYourBank()**: Accessor method for yourBank variable **// Nikhil**



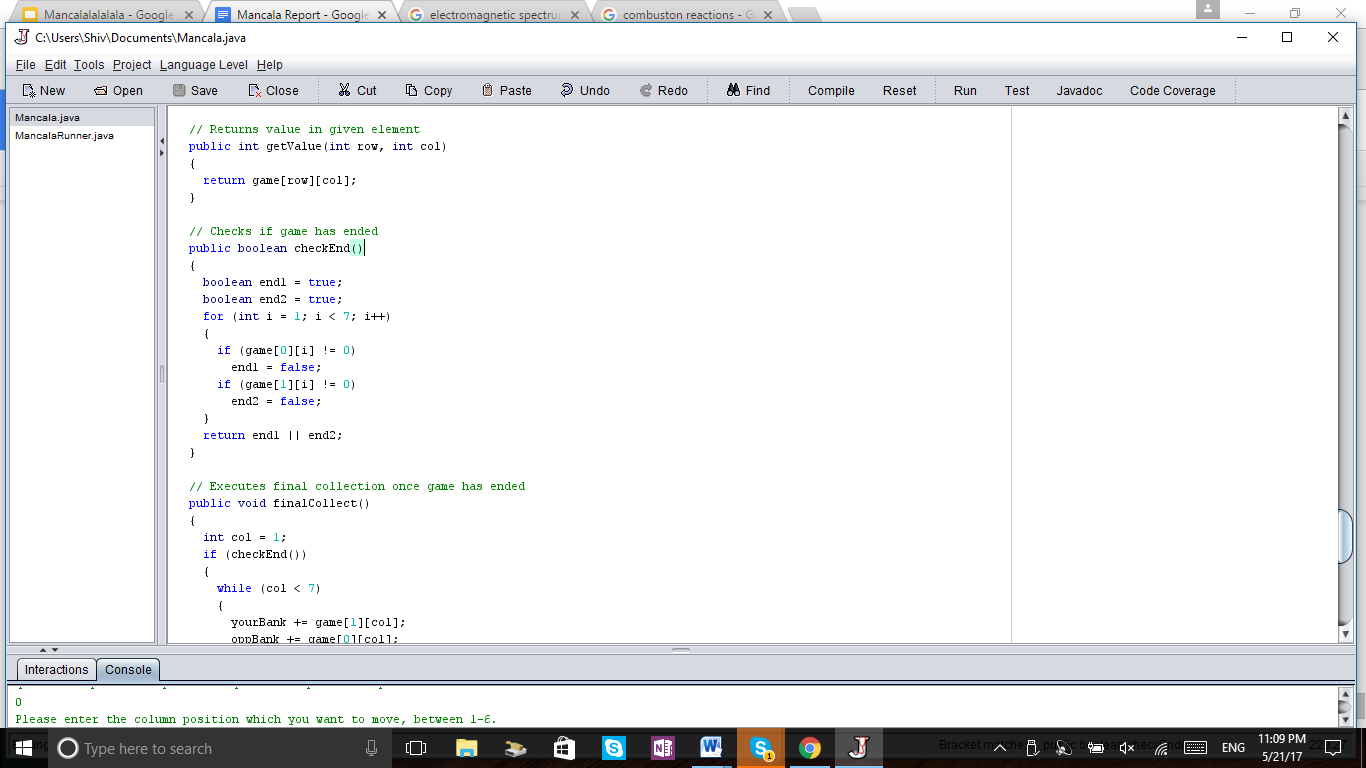
**getOppBank()**: Accessor method for oppBank variable **// Nikhil and Charlie**

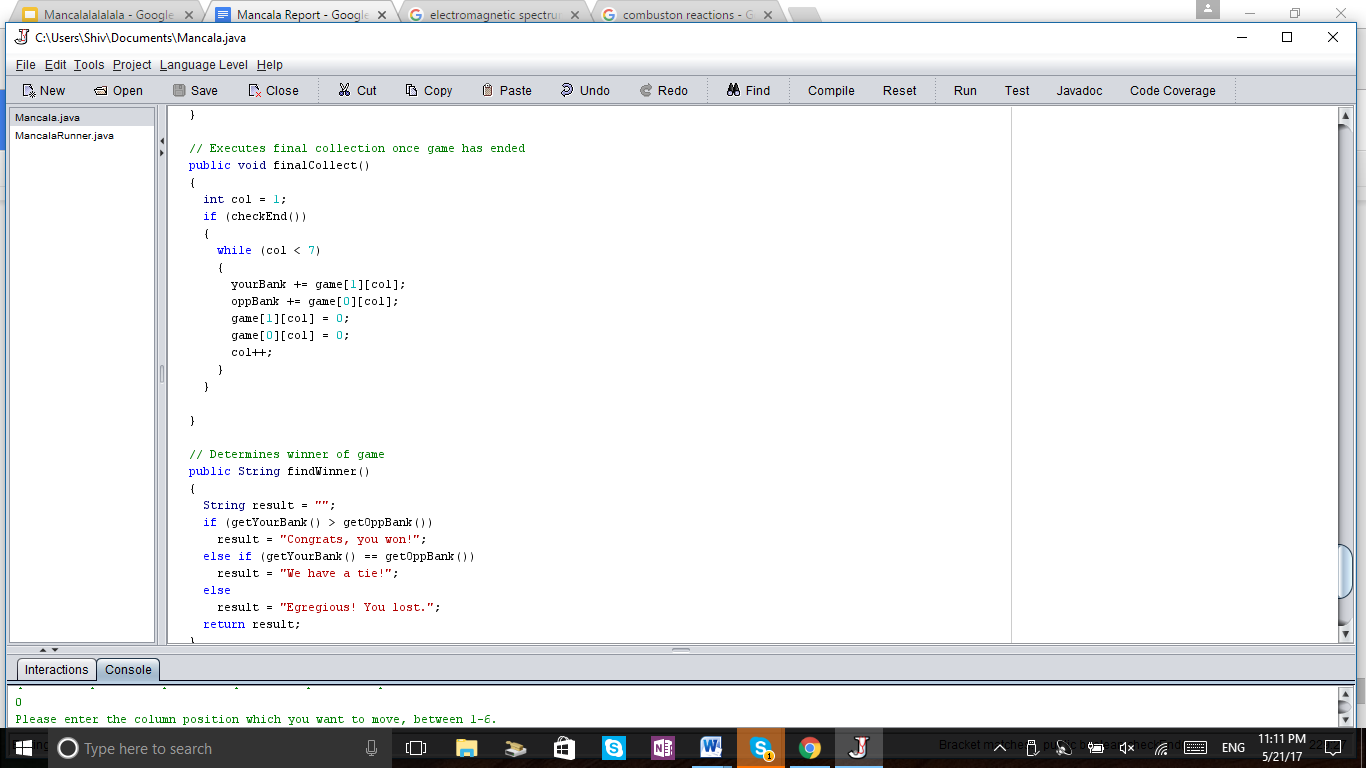


**getGoAgain():** Accessor method for goAgain variable **// Nikhil**

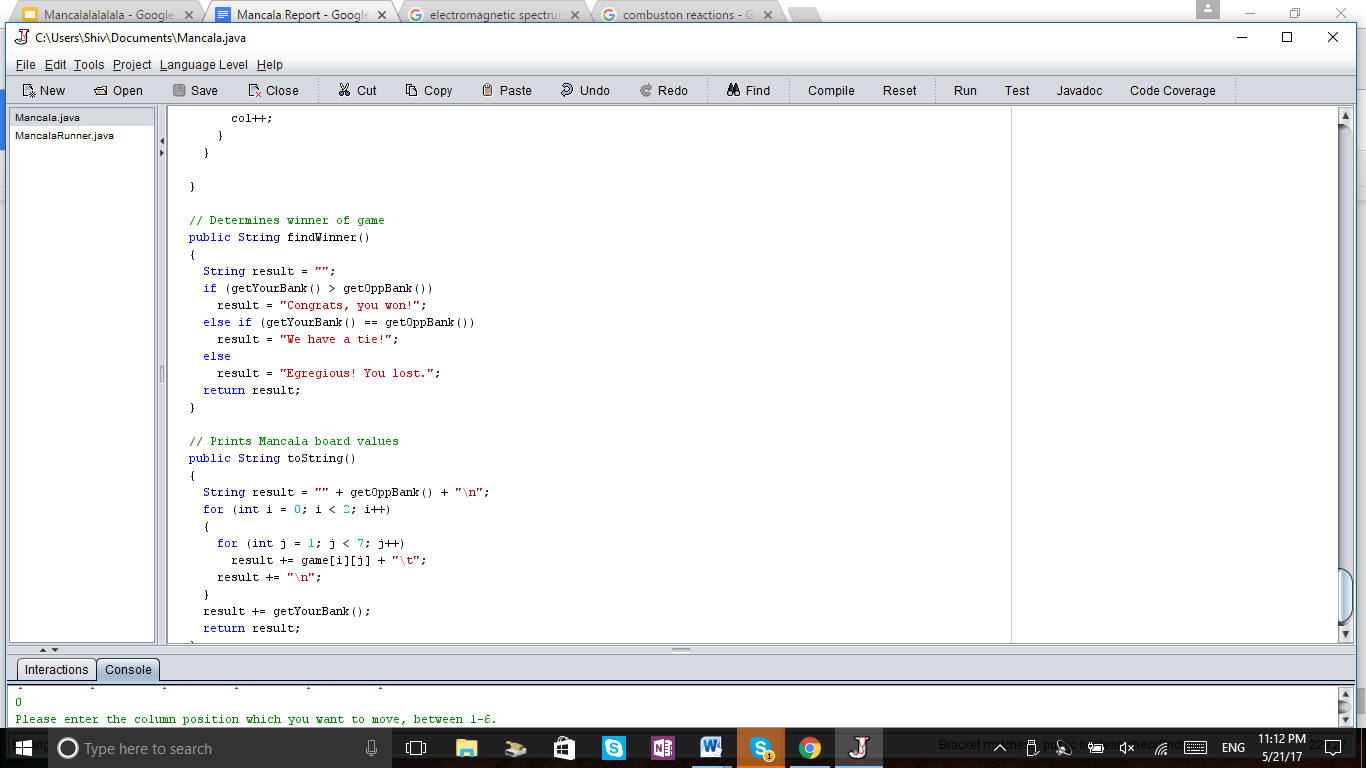


**checkEnd():** Checks if all of one row is empty to see if the game has ended **// Nikhil**

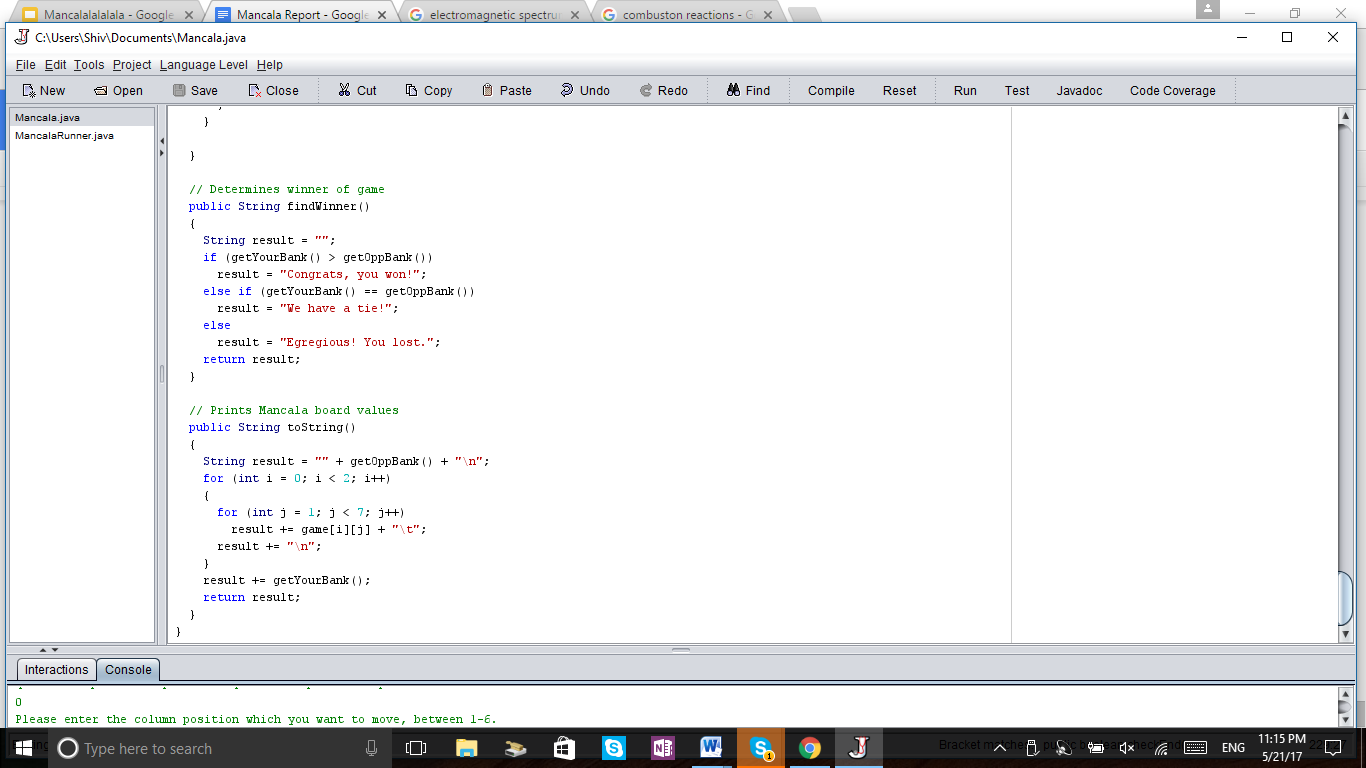


**finalCollect()**: This executes after checkEnd() and allows for the remaining pebbles to be placed 

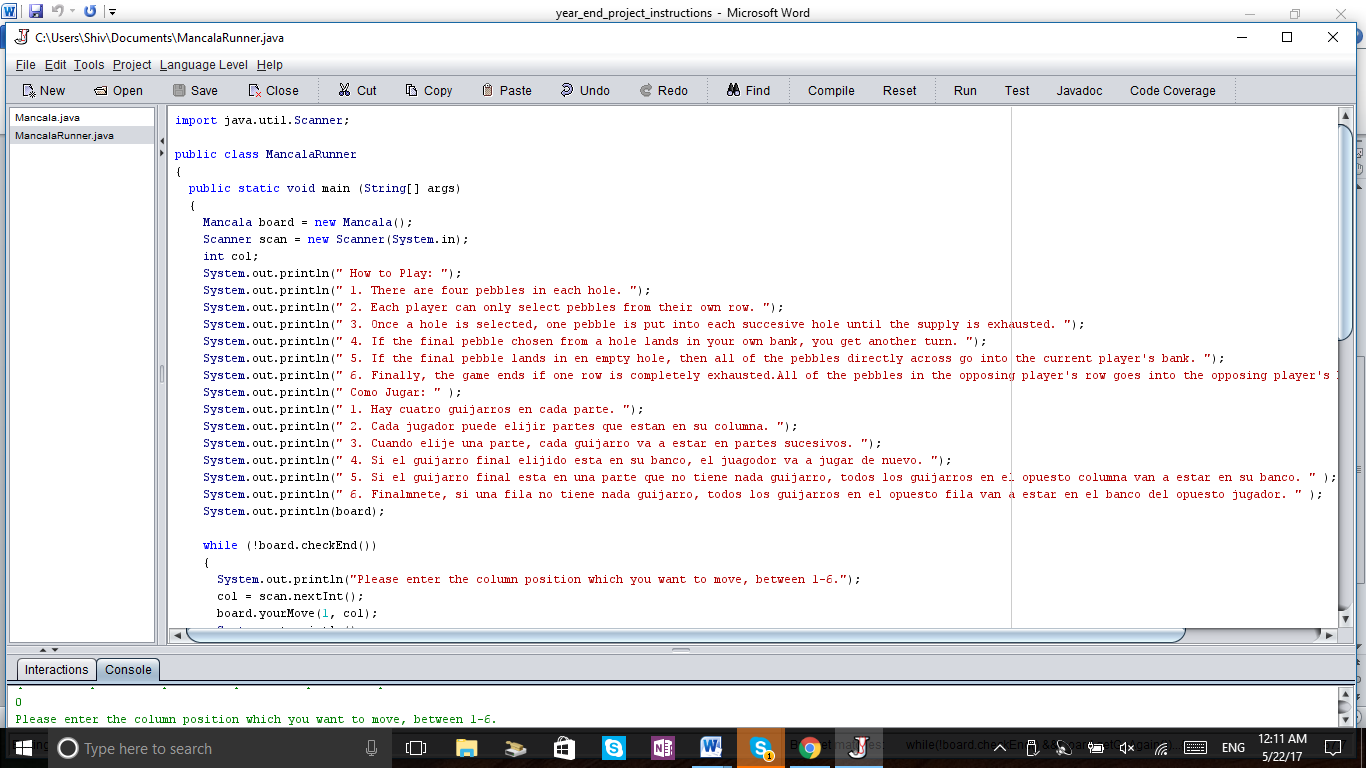
into one’s bank deservingly. **// Nikhil**

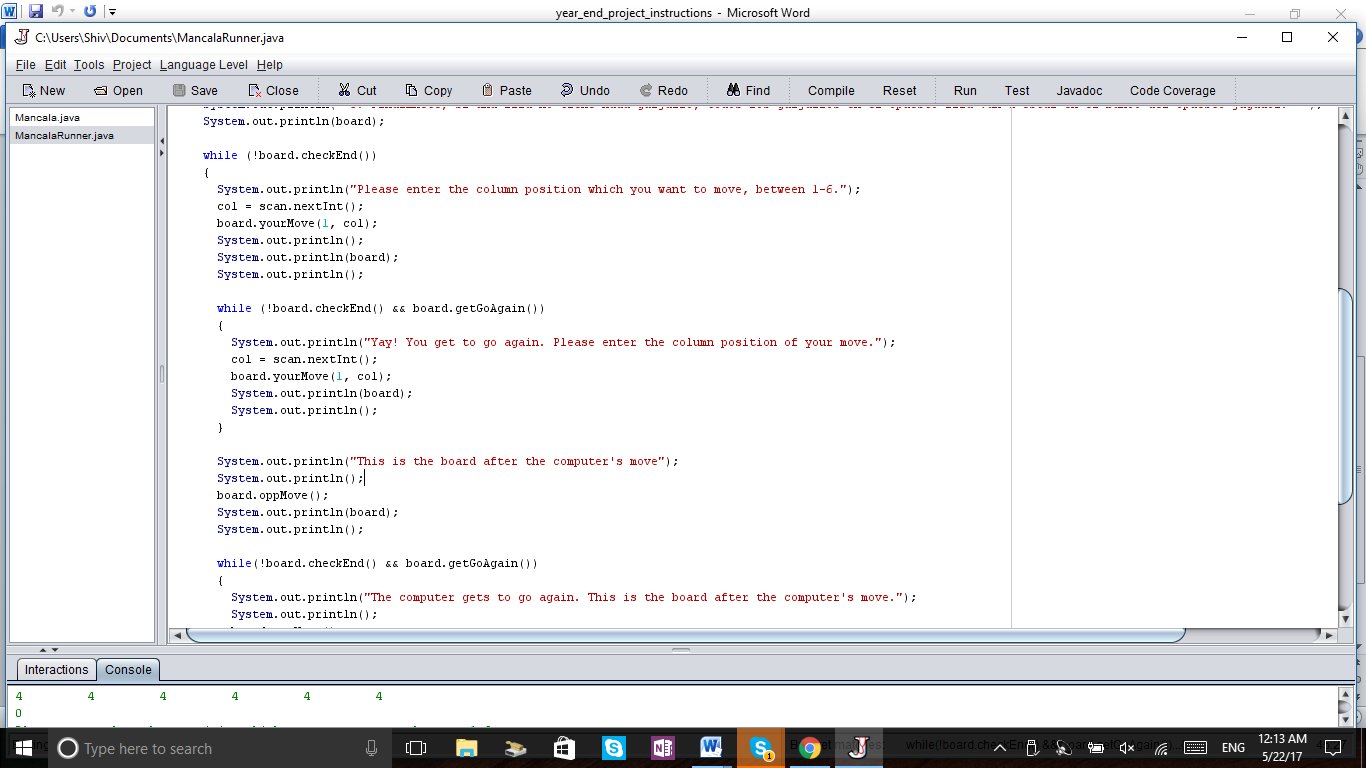
**findWinner():** This executes after finalCollect() and this determines the winner and prints out a message based off the number of pebbles in each bank. **// Nikhil** 

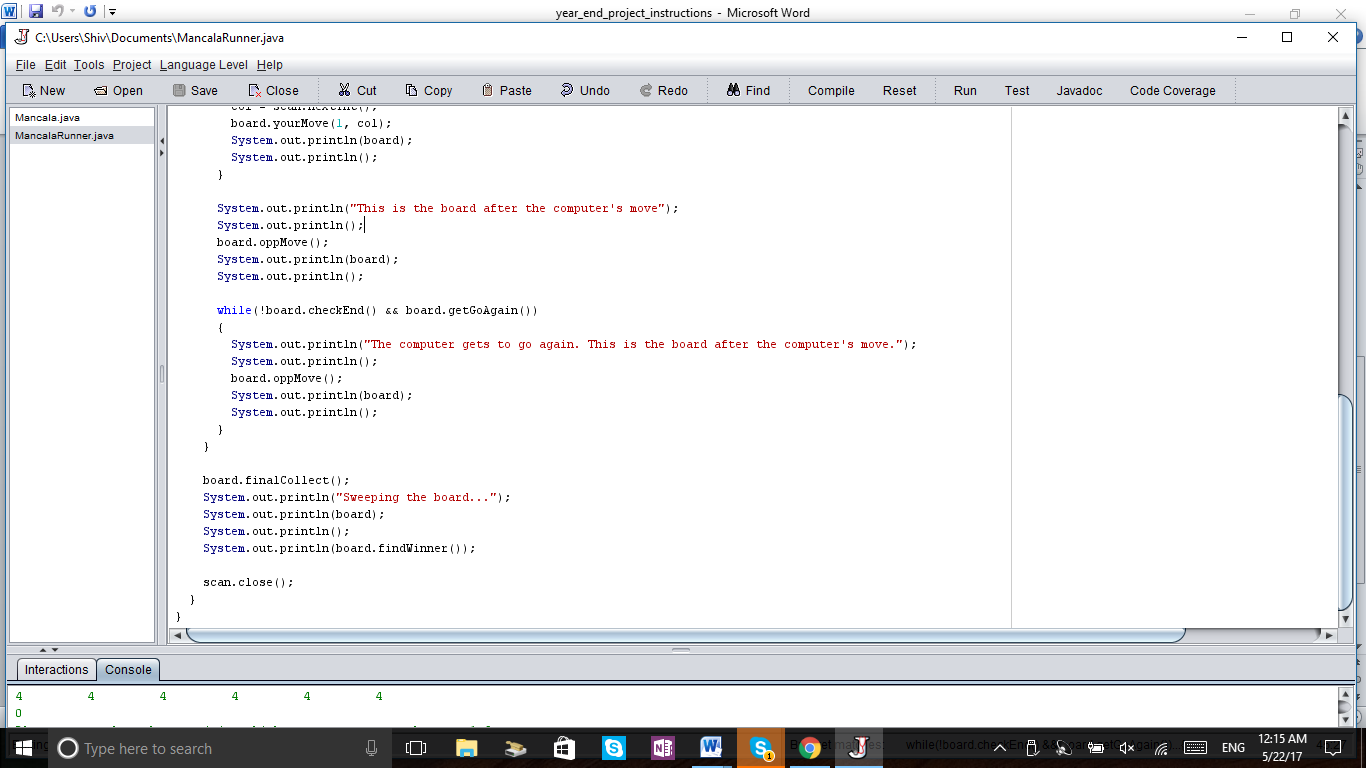
**toString():** Prints out the current status of the Mancala board. **// Nikhil and Charlie**



**Runner File Components // Nikhil**

Using Scanner, we were able to take the user’s input and make this an interactive game in the interactive panel. With while loops, we had the code loop between the user and computer’s move, and using checkEnd(), kept looping between the two until one row was completely empty. After every move, the code prints out the state of the board and banks with toString(). Using getGoAgain(), we had the code check if the players move ended in their own bank and had the players go again if so. Once a player’s row was completely empty, using finalCollect(), the code collected all the remaining pebbles and deposited them into their corresponding banks. From there, using findWinner(), the code determined the winner and printed a message accordingly.





**Original Code- Class File // Charlie**

public class Mancala  
{  
 private int[][] game = new int [2][8];  
 private int yourBank = 0;  
 private int oppBank = 0;  
 private boolean goAgain = false;  
   
   
 // Creates Mancala board with 4 stones in each element  
 public Mancala ()  
 {  
 for (int i = 0; i < game.length; i++)  
 for (int j = 1; j < 7; j++)  
 game[i][j] = 4;   
 }  
   
 // Executes player move based on input  
 // Precondition: row is always 1  
 public void yourMove(int row, int col)  
 {  
 int spaces = game[row][col];  
 boolean go = true;  
 int start = col + 1;  
 goAgain = false;  
   
 // Outer loop allows for move continuation if needed  
 while (spaces > 0)  
 {  
 if (go)  
 {  
 game[row][col] = 0;  
 go = false;  
 }  
  
 // Starts on bottom row  
 while (start < 7 && spaces > 0)  
 {  
 game[row][start]++;  
 spaces--;  
 start++;  
 }   
   
 // Deposits stone into player's bank  
 // Checks if player can go again  
 if (spaces > 0)  
 {  
 yourBank++;  
 spaces--;  
 if (spaces == 0)  
 goAgain = true;  
 else  
 goAgain = false;  
 }  
   
 // Adjusts parameters to opp's row  
 if (spaces > 0)  
 {  
 row = 0;  
 start = 6;  
 }  
   
 // Loops through opp's row  
 while (start > 0 && spaces > 0)  
 {  
 game[row][start]++;  
 start--;  
 spaces--;  
 }  
   
 // Adjusts parameters back to own row if needed  
 if (spaces > 0)  
 {  
 row = 1;  
 start = 1;  
 }  
 }  
   
 // Doesn't run through code if move ends in bank  
 if (!goAgain)  
 {  
 // Adjusts parameters for additional stone collection  
 if (row == 0 && start == 7)  
 row = 1;  
 else if (row == 1 && start == 1)  
 row = 0;  
 else if (row == 1)  
 start--;  
 else  
 start++;  
   
   
 // Checks and executes additional stone collection  
 if (game[row][start] == 1)  
 {  
 if (row == 0)  
 yourBank += game[1][start] + 1;  
 else  
 yourBank += game[0][start] + 1;  
 game[0][start] = 0;  
 game[1][start] = 0;  
 }  
 }  
 }   
   
 // Executes Opp move based on Math.random()  
 // Precondition: row is always 0  
 public void oppMove()  
 {  
 int row = 0;  
 int col = (int)(Math.random()\*6 + 1);  
 int spaces = game[row][col];  
 goAgain = false;  
   
 // Ensures opp doesn't choose empty slot  
 while (spaces == 0)  
 {  
 col = (int)(Math.random()\*6 + 1);  
 spaces = game[row][col];  
 }  
   
 boolean go = true;  
 int start = col - 1;  
   
 while (spaces > 0)  
 {  
 if (go)  
 {  
 game[row][col] = 0;  
 go = false;  
 }  
   
 // Starts on opp's row  
 while (start > 0 && spaces > 0)  
 {  
 game[row][start]++;  
 start--;  
 spaces--;  
 }  
   
 // Deposits stone into opp's bank  
 // Checks if opp can go again  
 if (spaces > 0)  
 {  
 oppBank++;  
 spaces--;  
 if (spaces == 0)  
 goAgain = true;  
 else  
 goAgain = false;  
 }  
  
   
 // Adjusts parameters to player's row  
 if (spaces > 0)  
 {  
 row = 1;  
 start = 1;  
 }  
   
 // Loops through player's row  
 while (start < 7 && spaces > 0)  
 {  
 game[row][start]++;  
 spaces--;  
 start++;  
 }  
   
 // Adjusts parameters back to opp's row if needed  
 if (spaces > 0)  
 {  
 row = 0;  
 start = 6;  
 }  
 }  
   
 // Doesn't run through code if move ends in bank  
 if (!goAgain)  
 {  
 // Adjusts parameters for additional stone collection  
 if (row == 0 && start == 7)  
 row = 1;  
 else if (row == 1 && start == 1)  
 row = 0;  
 else if (row == 1)  
 start--;  
 else  
 start++;  
   
   
 // Checks and executes additional stone collection  
 if (game[row][start] == 1)  
 {  
 if (row == 0)  
 oppBank += game[1][start] + 1;  
 else  
 oppBank += game[0][start] + 1;  
 game[0][start] = 0;  
 game[1][start] = 0;  
 }  
 }  
 }  
   
   
 // Accesses player's bank value   
 public int getYourBank()  
 {  
 return yourBank;  
 }  
   
 // Accesses opp's bank value  
 public int getOppBank()  
 {  
 return oppBank;  
 }  
   
 // Accesses if player or opp can go again  
 public boolean getGoAgain()  
 {  
 return goAgain;  
 }  
   
 // Returns value in given element  
 public int getValue(int row, int col)  
 {  
 return game[row][col];  
 }  
   
 // Checks if game has ended  
 public boolean checkEnd()  
 {  
 boolean end1 = true;  
 boolean end2 = true;  
 for (int i = 1; i < 7; i++)  
 {  
 if (game[0][i] != 0)  
 end1 = false;  
 if (game[1][i] != 0)  
 end2 = false;  
 }  
 return end1 || end2;  
 }  
   
 // Executes final collection once game has ended  
 public void finalCollect()  
 {  
 int col = 1;  
 if (checkEnd())  
 {  
 while (col < 7)  
 {  
 yourBank += game[1][col];  
 oppBank += game[0][col];  
 game[1][col] = 0;  
 game[0][col] = 0;  
 col++;  
 }  
 }  
   
 }  
   
 // Determines winner of game  
 public String findWinner()  
 {  
 String result = "";  
 if (getYourBank() > getOppBank())  
 result = "Congrats, you won!";  
 else if (getYourBank() == getOppBank())  
 result = "We have a tie!";  
 else  
 result = "Egregious! You lost.";  
 return result;  
 }  
   
 // Prints Mancala board values  
 public String toString()  
 {  
 String result = "" + getOppBank() + "\n";  
 for (int i = 0; i < 2; i++)  
 {  
 for (int j = 1; j < 7; j++)  
 result += game[i][j] + "\t";  
 result += "\n";  
 }  
 result += getYourBank();  
 return result;  
 }  
}

**Original Code- Runner Program // Charlie**

import java.util.Scanner;

public class MancalaRunner

{

public static void main (String[] args)

{

Mancala board = new Mancala();

Scanner scan = new Scanner(System.in);

int col;

System.out.println(" How to Play: ");

System.out.println(" 1. There are four pebbles in each hole. ");

System.out.println(" 2. Each player can only select pebbles from their own row. ");

System.out.println(" 3. Once a hole is selected, one pebble is put into each successive hole until the supply is exhausted. ");

System.out.println(" 4. If the final pebble chosen from a hole lands in your own bank, you get another turn. ");

System.out.println(" 5. If the final pebble lands in en empty hole, then all of the pebbles directly across go into the current player's bank. ");

System.out.println(" 6. Finally, the game ends if one row is completely exhausted. All of the pebbles in the opposing player's row goes into the opposing player's bank ");

System.out.println(" Como Jugar: " );

System.out.println(" 1. Hay cuatro guijarros en cada parte. ");

System.out.println(" 2. Cada jugador puede elijir partes que estan en su columna. ");

System.out.println(" 3. Cuando elije una parte, cada guijarro va a estar en partes sucesivos. ");

System.out.println(" 4. Si el guijarro final elijido esta en su banco, el juagodor va a jugar de nuevo. ");

System.out.println(" 5. Si el guijarro final esta en una parte que no tiene nada guijarro, todos los guijarros en el opuesto columna van a estar en su banco. " );

System.out.println(" 6. Finalmnete, si una fila no tiene nada guijarro, todos los guijarros en el opuesto fila van a estar en el banco del opuesto jugador. " );

System.out.println(board);

while (!board.checkEnd())

{

System.out.println("Please enter the column position which you want to move, between 1-6.");

col = scan.nextInt();

board.yourMove(1, col);

System.out.println();

System.out.println(board);

System.out.println();

while (!board.checkEnd() && board.getGoAgain())

{

System.out.println("Yay! You get to go again. Please enter the column position of your move.");

col = scan.nextInt();

board.yourMove(1, col);

System.out.println(board);

System.out.println();

}

System.out.println("This is the board after the computer's move");

System.out.println();

board.oppMove();

System.out.println(board);

System.out.println();

while(!board.checkEnd() && board.getGoAgain())

{

System.out.println("The computer gets to go again. This is the board after the computer's move.");

System.out.println();

board.oppMove();

System.out.println(board);

System.out.println();

}

}

board.finalCollect();

System.out.println("Sweeping the board...");

System.out.println(board);

System.out.println();

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

scan.close();

}

}

**Troubleshooting Process // Nikhil**

In troubleshooting our problems, we tested our program by manually playing against the computer, and matching the java game board with a real life game board after every move. From there, we had a physical representation of what the game should actually look like and were able to compare that to what our program had done.

**Conclusion // Charlie and Nikhil**

We learned that in creating a project in this size and scope that it is imperative to spend enough time in the design phase before jumping onto implementation. When we first started, we did not realize there were so many moving parts that had to be accounted for in trying to keep compliance with all the rules, and soon found ourselves constantly going back to the drawing board on how to properly execute certain moves. In this project and in the real world as a whole, organization of thoughts and ideas is key to successfully completing any assignment, and as always, Mr. Hatcher was 100% right to us. Even though we may thought we had covered every scenario, the only way to check that your code is foolproof is by constant testing of its execution-to see if the code can handle all situations from all angles, and from there, that would give us an idea of where to improve and revise our code. As a result, the lessons learned here were really more lessons relearned-that the bulk of computer science work is in the design phase and you need to be strategic and methodical in your code testing.