```
////////////////////// ALL ASSIGNMENTS INCLUDE THIS SECTION ///////////////////
//
// Title: BP1
// Files: Battleship.java
// Course: CS 200 Spring 2018
//
// Author: Jackson Hellmers
// Email: jhellmers@wisc.edu
// Lecturer's Name: Marc R
//
///////////////////////////// CREDIT OUTSIDE HELP /////////////////////////////
//
// Students who get help from sources other than their partner must fully
// acknowledge and credit those sources of help here. Instructors and TAs do
// not need to be credited here, but tutors, friends, relatives, room mates
// strangers, etc do. If you received no outside help from either type of
// source, then please explicitly indicate NONE.
//
// Persons: (identify each person and describe their help in detail)
// Online Sources: (identify each URL and describe their assistance in detail)
//
////////////////////////////////// 80 COLUMNS WIDE ////////////////////////////////


import java.lang.Math;
import java.util.Scanner;
import java.util.Random;

/*
 * The comments above each method give a brief summary of its purpose.
 */

public class Battleship {

    /*
     * This method takes a string and treats the string as a base 26 number
(where A=0, B=1...).
     * I first filled an array with each element corresponding to the next
letter in the string.
     * Then, using a for loop I found the value of each letter at its index and
added it to a total
     * sum. The method then returns the sum once every character has be added
up.
     */
    public static int coordAlphaToNum(String coord) {
        coord = coord.toUpperCase();
        int i = 0;
        char[] characters = new char[coord.length()];
        characters = coord.toCharArray();
        for (int j = 0; j < characters.length; j++) {
            i = i + ((int) characters[characters.length - 1 - j] - 65) * (int)
Math.pow(26, j);
        }
        return i;
    }

    /*
     * The first portion of this method determines the highest index of the
number in its string
     * form by seeing how many times it can be divided by 26. It then takes
advantage of the integer
     * division and modulo operations to find the value at each index. Finally
each calculated value
     * is added to 65 as this will give it the corresponding ASCII code to which
```

```
character it
     * represents.
     */
    public static String coordNumToAlpha(int coord) {
        char addToString = '\0';
        int res = coord;
        int highestPow = 0;
        String str = "";
        while (coord >= 26) {
            coord = coord / 26;
            highestPow += 1;
        }
        coord = res;
        for (int i = 0; i <= highestPow; i++) {
            res = coord / (int) (Math.pow(26, highestPow - i));
            coord = coord - (res * (int) (Math.pow(26, highestPow - i)));
            addToString = (char) (res + 65);
            str = str + addToString;
        }
        return str;

    }

    /*
     * Prompts the user to enter an integer between the given maximum and
minimum values. If the
     * input is not between the max or min the user is told their input is
invalid and is prompted
     * again until their input satisfies the max and min conditions.
     */
    public static int promptInt(Scanner sc, String valName, int min, int max) {
        System.out.print("Enter the " + valName + " (" + min + " to " + max +
"): ");
        int input = sc.nextInt();
        while (input < min || input > max) {
            System.out.println("Invalid value.");
            System.out.print("Enter the " + valName + " (" + min + " to " + max
+ "): ");
            input = sc.nextInt();
        }
        return input;
    }

    /*
     * This method prompts the user to enter a string. However, if the string is
not between the
     * maximum and minimum string values allowed the phrase "Invalid value." is
printed out and
     * the prompt is run again. The max and min values are based upon how the
input compares
     * alphabetically (case does not matter).
     */
    public static String promptStr(Scanner sc, String valName, String min,
String max) {
        String input = "";
        System.out.print("Enter the " + valName + " (" + min + " to " + max +
"): ");
        input = sc.next();
        input = input.trim().toUpperCase();
        while ((input.compareTo(min) < 0) || (input.compareTo(max) > 0)) {
            System.out.println("Invalid value.");
            System.out.print("Enter the " + valName + " (" + min + " to " + max
+ "): ");
            input = sc.next();
```

```
            input = input.trim().toUpperCase();
        }
        return input;
    }

    /*
     * Prompts the user to enter a character. If an entire string is input, then
the function just
     * treats the first character as the input. If the string is empty, the
method treats it as the
     * null character. All whitespace is trimmed and case does not matter.
     */
    public static char promptChar(Scanner sc, String prompt) {
        char select = 'a';
        String input = "";
        System.out.print(prompt);
        input = sc.next();
        input = input.trim().toLowerCase();
        if (input.isEmpty()) {
            return '\0';
        } else {
            select = input.charAt(0);
            return select;
        }
    }

    /*
     * Runs nested for loops to completely fill every element of the board array
with the default
     * water character '~'.
     */
    public static void initBoard(char board[][]) {
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[i].length; j++) {
                board[i][j] = Config.WATER_CHAR;
            }
        }
    }

    /*
     * The method starts by printing out the first row, consisting of the
character representation
     * of its numerical value. This is done by calling the method
"coordNumToAlpha" which the num
     * is the column value. It then begins to print the rows and columns of the
2D array, with each
     * row starting with its array index value.
     */
    public static void printBoard(char board[][], String caption) {
        String column = "";
        System.out.println(caption + ":");
        for (int i = 0; i < Config.MAX_COL_WIDTH; i++) {
            System.out.print(" ");
        }
        for (int l = 0; l < board[0].length; l++) {
            column = coordNumToAlpha(l);
            for (int i = 0; i < Config.MAX_COL_WIDTH - column.length(); i++) {
                System.out.print(" ");
            }
            System.out.print(column);
        }
        System.out.println("");
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[i].length; j++) {
```

```
                if (j == 0) {
                    int length = String.valueOf(i).length();
                    for (int k = 0; k < Config.MAX_COL_WIDTH - length; k++) {
                        System.out.print(" ");
                    }
                    System.out.print(i);
                }
                for (int l = 0; l < Config.MAX_COL_WIDTH - 1; l++) {
                    System.out.print(" ");
                }
                System.out.print(board[i][j]);
            }
            System.out.println("\n");
        }
    }

    /*
     * The method checks to see if a ship can be placed in a certain area given
its coordinates,
     * length, and direction. An integer counter is used within a for loop that
will either run
     * through every x-coordinate or every y-coordinate (depending on direction)
and will increment
     * every time the checked coordinate is filled by the water character. At
the end of the code
     * if the counter is equal to the length of the ship, then it is known every
element is open
     * and the ship can be properly placed and the method returns a 1. If every
element is not open
     * then a -1 is returned and if the ship goes off the board a -2 is
returned.
     */
    public static int checkWater(char board[][], int xcoord, int ycoord, int
len, boolean dir) {
        int count = 0;

        if (dir == true) {
            if (ycoord + len > board.length) {
                return -2;
            } else {
                for (int i = 0; i < len; i++) {
                    if (board[ycoord + i][xcoord] == Config.WATER_CHAR) {
                        count += 1;
                    }
                }
            }
        } else {
            if (xcoord + len > board[ycoord].length) {
                return -2;
            } else {
                for (int i = 0; i < len; i++) {
                    if (board[ycoord][xcoord + i] == Config.WATER_CHAR) {
                        count += 1;
                    }
                }
            }
        }
        if (count == len) {
            return 1;
        } else {
            return -1;
        }
    }
```

```
    /*
     * Checks to see if any ship IDs remain. If any do exist then all ships are
not sunk and the
     * boolean false is returned. If all ships are sunk then true is returned.
     */
    public static boolean checkLost(char board[][]) {
        int idCounter = 0;
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                if (board[i][j] != Config.WATER_CHAR && board[i][j] !=
Config.HIT_CHAR
                    && board[i][j] != Config.MISS_CHAR) {
                    idCounter += 1;
                }
            }
        }
        return idCounter == 0;
    }

    /*
     * This method will update the board in the parameters with a ship by
replacing the water
     * characters at its location with a number that corresponds to its id. The
method does this
     * by first finding which direction the ship faces and then increments as
many times as the
     * ship is long in that same direction. If the ship is properly placed the
method returns true.
     * However, if for some reason the ship cannot be placed, the method returns
false;
     */
    public static boolean placeShip(char board[][], int xcoord, int ycoord, int
len, boolean dir,
        int id) {
        if (dir == true) {
            if (ycoord + len > board.length) {
                return false;
            }
            for (int i = 0; i < len; i++) {
                for (int j = 0; j < 1; j++) {
                    board[ycoord + i][xcoord] = (char) (id + 48); // Add 48 to
get ASCII number char
                }
            }
            return true;
        } else  {
            if (xcoord + len > board[ycoord].length) {
                return false;
            }
            for (int i = 0; i < len; i++) {
                for (int j = 0; j < len; j++) {
                    board[ycoord][xcoord + i] = (char) (id + 48); // Add 48 to
get ASCII number char
                }
            }
            return true;
        }
    }

    /*
     * Using a random number generator to produce 2 int values (representing the
x and y coordinates
     * and 1 double value (to determine the ships direction), the method tries a
maximum of 20
```

```
     * times to try and place a ship the same length as the user just did. If
the ship is unable
     * to be placed within those 20 tries the method returns the boolean value
false.
     */
    public static boolean placeRandomShip(char board[][], int len, int id,
Random rand) {
        boolean dir;
        int count = 0;
        int xcoord;
        int ycoord;
        while (count < Config.RAND_SHIP_TRIES) {
            dir = rand.nextBoolean();
            if (dir == true) {
                xcoord = rand.nextInt(board[0].length);
                ycoord = rand.nextInt(board.length - len + 1);
            } else {
                xcoord = rand.nextInt(board[0].length - len + 1);
                ycoord = rand.nextInt(board.length);
            }
            if (checkWater(board, xcoord, ycoord, len, dir) == 1) {
                placeShip(board, xcoord, ycoord, len, dir, id);
                return true;
            }
            count += 1;
        }
        return false;
    }

    /*
     * This method begins with a while that will automatically run its first
iteration, as the
     * repeater char is defaulted to 'y'. Once in the while loop, the user is
first prompted which
     * direction they want their ship to be placed. Next, the user is asked to
enter the desired
     * length of their ship Then the user is also prompted as to what they want
the upper-left
     * coordinate to be. The max and min values the user can enter for each axis
are calculated by
     * taking the ships length and direction into account. Once this has all
been completed, the
     * method checks to see if the ship can be placed onto the desired
coordinate range. If it can,
     * it will do so and then attempt to place a random computer ship with the
same length. If
     * either the user or computer ship cannot be placed the method will return
false. If both are
     * placed, the method returns true.
     */
    public static boolean addShip(Scanner sc, char boardPrime[][], char
boardOpp[][], int id,
        Random rand) {
        boolean dir;
        boolean canPlaceOpp;
        char orientation = '\0';
        char repeater = 'y';
        int canPlace;
        int maxLength;
        int len;
        String xcoord;
        int ycoord;
        String input = "";
        while (repeater == 'y') {
```

```
            printBoard(boardPrime, "My Ships");
            System.out.print("Vertical or horizontal? (v/h): ");
            input = sc.next();
            input = input.toLowerCase();
            orientation = input.charAt(0);
            if (orientation == 'v') {
                dir = true;
                maxLength = boardPrime.length;
                len = promptInt(sc, "ship length", Config.MIN_SHIP_LEN,
maxLength);
                xcoord = promptStr(sc, "x-coord", coordNumToAlpha(0),
                    coordNumToAlpha(boardPrime[0].length - 1));
                ycoord = promptInt(sc, "y-coord", 0, boardPrime.length - len);
                canPlace = checkWater(boardPrime, coordAlphaToNum(xcoord),
ycoord, len, dir);
            } else {
                dir = false;
                maxLength = boardPrime[0].length;
                len = promptInt(sc, "ship length", Config.MIN_SHIP_LEN,
maxLength);
                xcoord = promptStr(sc, "x-coord", coordNumToAlpha(0),
                    coordNumToAlpha(boardPrime[0].length - len));
                ycoord = promptInt(sc, "y-coord", 0, boardPrime.length - 1);
                canPlace = checkWater(boardPrime, coordAlphaToNum(xcoord),
ycoord, len, dir);
            }
            if (canPlace == 1) {
                placeShip(boardPrime, coordAlphaToNum(xcoord), ycoord, len, dir,
id);
                if (placeRandomShip(boardOpp, len, id, rand) == false) {
                    System.out.println("Unable to place opponent ship: " + id);
                    return false;
                }
                return true;
            }
            repeater = promptChar(sc, "No room for ship. Try again? (y/n): ");
        }
        return false;
    }

    /*
     * The method first checks to see if the shot coordinate is on the board. If
it is not, then
     * the int "-1" is returned. If it is on the board, the method then checks
to see if the char
     * in the corresponding array element is a water char. If that is true, then
the method returns
     * the int "2". If both of the first two conditions are passed, then the
method finally checks
     * if the array element is either a hit or miss char. If either of these are
true then the int
     * "3" is returned. If none of the prior conditions are met, then the method
will return the
     * int "1", signifying that shot will hit a ship.
     */
    public static int takeShot(char[][] board, int x, int y) {
        if (board.length <= y || board[0].length <= x) {
            return -1;
        } else if (board[y][x] == Config.WATER_CHAR) {
            return 2;
        } else if (board[y][x] == Config.MISS_CHAR || board[y][x] ==
Config.HIT_CHAR) {
            return 3;
        } else {
```

```
            return 1;
        }
    }

    /*
     * This method is used to allow the user to enter which coordinate they
would like to fire
     * upon. It begins by prompting the user to enter the X character coordinate
and Y number
     * coordinate they desire. It will then use the takeShot method to check the
entered
     * coordinates. If anything other than a 1 or 2 are returned the method will
enter a while loop.
     * The while loop checks to see what was returned from takeShot and will
print a corresponding
     * depending on if it was a -1 or a 3. If -1, "Coordinates out-of-bounds!"
is printed. If 3,
     * "Shot location previously targeted!" is printed. It will then ask the
user to enter
     * a new set of coordinates. Once the while loop has ended, the method will
then update the
     * coordinate on the board with the correct character (depending on hit or
miss).
     */
    public static void shootPlayer(Scanner sc, char[][] board, char[][]
boardTrack) {
        int returnValue;
        String x =
            promptStr(sc, "x-coord shot", coordNumToAlpha(0),
coordNumToAlpha(board[0].length - 1));
        int xcoord = coordAlphaToNum(x);
        int ycoord = promptInt(sc, "y-coord shot", 0, board.length - 1);
        returnValue = takeShot(board, xcoord, ycoord);
        while (returnValue == -1 || returnValue == 3) {
            if (returnValue == -1) {
                System.out.println("Coordinates out-of-bounds!");
            } else if (returnValue == 3) {
                System.out.println("Shot location previously targeted!");
            }
            x = promptStr(sc, "x-coord shot", coordNumToAlpha(0),
                coordNumToAlpha(board[0].length - 1));
            xcoord = coordAlphaToNum(x);
            ycoord = promptInt(sc, "y-coord shot", 0, board.length - 1);
            returnValue = takeShot(board, xcoord, ycoord);
        }
        if (returnValue == 1) {
            board[ycoord][xcoord] = Config.HIT_CHAR;
            boardTrack[ycoord][xcoord] = Config.HIT_CHAR;
        } else if (returnValue == 2) {
            board[ycoord][xcoord] = Config.MISS_CHAR;
            boardTrack[ycoord][xcoord] = Config.MISS_CHAR;
        }
    }

    /*
     * This method begins by randomly generating a x-coordinate and y-coordinate
(each bounded by
     * the length of the board in that axis direction). It then uses the
takeShot() method to make
     * sure targeted coordinate is both on the board and hasn't already been
targeted and will
     * continue to generate new coordinates until these two conditions are true.
It will then
     * finally update the board with a hit or miss character at the designated
```

```
x,y coordinate
     * depending upon if a ship is present there or not.
     */
    public static void shootComputer(Random rand, char[][] board) {
        int xcoord = rand.nextInt(board[0].length);
        int ycoord = rand.nextInt(board.length);
        while (takeShot(board, xcoord, ycoord) == 3 || takeShot(board, xcoord,
ycoord) == -1) {
            xcoord = rand.nextInt(board[0].length);
            ycoord = rand.nextInt(board.length);
        }
        if (takeShot(board, xcoord, ycoord) == 1) {
            board[ycoord][xcoord] = Config.HIT_CHAR;
        } else if (takeShot(board, xcoord, ycoord) == 2) {
            board[ycoord][xcoord] = Config.MISS_CHAR;
        }
    }

    /**
     * This is the main method for the Battleship game. It consists of the main
game and play again
     * loops with calls to the various supporting methods. When the program
launches (prior to the
     * play again loop), a message of "Welcome to Battleship!", terminated by a
newline, is
     * displayed. After the play again loop terminiates, a message of "Thanks
for playing!",
     * terminated by a newline, is displayed.
     *
     * The Scanner object to read from System.in and the Random object with a
seed of Config.SEED
     * will be created in the main method and used as arguments for the
supporting methods as
     * required.
     *
     * Also, the main method will require 3 game boards to track the play: - One
for tracking the
     * ship placement of the user and the shots of the computer, called the
primary board with a
     * caption of "My Ship". - One for displaying the shots (hits and misses)
taken by the user,
     * called the tracking board with a caption of "My Shots"; and one for
tracking the ship
     * placement of the computer and the shots of the user. - The last board is
never displayed, but
     * is the primary board for the computer and is used to determine when a hit
or a miss occurs
     * and when all the ships of the computer have been sunk. Notes: - The size
of the game boards
     * are determined by the user input. - The game boards are 2d arrays that
are to be viewed as
     * row-major order. This means that the first dimension represents the y-
coordinate of the game
     * board (the rows) and the second dimension represents the x-coordinate
(the columns).
     *
     * @param args Unused.
     */
    public static void main(String[] args) {
        char repeater = 'y';
        Scanner sc = new Scanner(System.in);
        Random rand = new Random(Config.SEED);
        System.out.println("Welcome to Battleship!");
        while (repeater == 'y') {
```

```
            int height = promptInt(sc, "board height", Config.MIN_HEIGHT,
Config.MAX_HEIGHT);
            int width = promptInt(sc, "board width", Config.MIN_WIDTH,
Config.MAX_WIDTH);
            System.out.println("");
            char board[][] = new char[height][width];      // Creates primary
board
            char boardShots[][] = new char[height][width]; // Creates shots
taken board
            char boardOpp[][] = new char[height][width];   // Create opponents
board
            initBoard(board);
            initBoard(boardShots);
            initBoard(boardOpp);
            int numShips = promptInt(sc, "number of ships", Config.MIN_SHIPS,
Config.MAX_SHIPS);
            for (int i = 1; i <= numShips; i++) {
                addShip(sc, board, boardOpp, i, rand);
            }
            boolean userWin = false;
            boolean compWin = false;
            while (userWin == false && compWin == false) { // Checks if anyone
has won game
                printBoard(board, "My Ships");
                printBoard(boardShots, "My Shots");
                shootPlayer(sc, boardOpp, boardShots);
                userWin = checkLost(boardOpp);
                if (userWin == true) {
                    continue;                               /* If users shot
makes them win the while
                                                             * loop instantly
ends. As the continue
                                                             * will cause the
loop recheck conditions
                                                             */
                }
                shootComputer(rand, board);
                compWin = checkLost(board);
            }
            if (userWin == true) {
                System.out.println("Congratulations, you sunk all the computer's
ships!");
            } else if (compWin == true) {
                System.out.println("Oh no! The computer sunk all your ships!");
            }
            printBoard(board, "My Ships");
            printBoard(boardShots, "My Shots");
            repeater = promptChar(sc, "Would you like to play again? (y/n): ");
        }
        System.out.println("Thanks for playing!");
    }
}
```