Fall, 2020 CIT 591

Homework 8

This homework deals with 2 main topics

* Abstract classes, Polymorphism. Abstraction of a game (BattleShip)

* Recursion

You can work on this assignment with a partner. We will enable group submission on Gradescope.

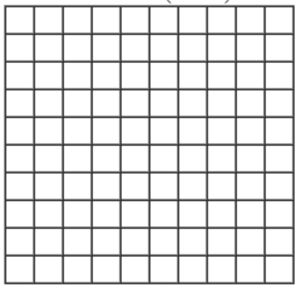
Part 1 - Battleship game

Overall idea

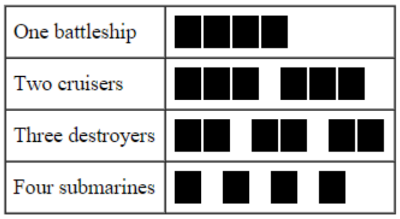
We are going to show you how to build a simple (only because there is no graphical user interface - GUI) version of the classic game Battleship. Battleship is usually a two-player game, where each player has a fleet and an ocean (hidden from the other player), and tries to be the first to sink the other player's fleet. We will doing just a solo version, where the computer places the ships, and the human attempts to sink them. We'll play this game on a 10x10 ocean with the following number of ships of each type.

You can play the game online over here https://battleship-game.org/en

The Ocean (10x10)



The Fleet



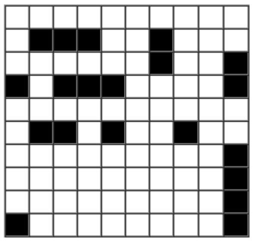
One battleship	■■■■
Two cruisers	■■■ ■■■
Three destroyers	■■ ■■ ■■
Four submarines	■ ■ ■ ■

How to play

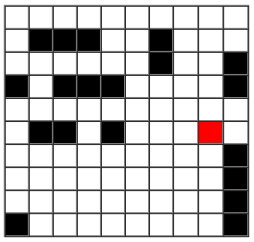
Please take a look at these rules even if you have played Battleship before in your life. Remember this is Human v Computer.

The computer places the ten ships on the ocean in such a way that no ships are immediately adjacent to each other, either horizontally, vertically, or diagonally. Take a look at the following diagrams for examples of legal and illegal placements (we have not shown a diagram of vertically adjacent but hopefully you get the idea).

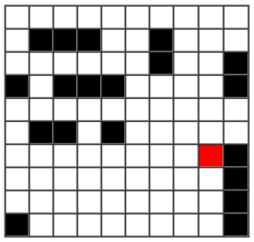
Legal arrangement



Illegal--ships diagonally adjacent



Illegal--ships horizontally adjacent



The human player does not know where the ships are. The initial display of the ocean shows a 10 by 10 array of locations, all the same. The human player tries to hit the ships, by calling out a row and column number. The computer responds with one bit of information saying ”hit” or ”miss.” When a ship is hit but not sunk, the program does not provide any information about what kind of a ship was hit. However, when a ship is hit and sinks, the program prints out a message ”You just sank a ship-type.” After each shot, the computer re-displays the ocean with the new information. A ship is ”sunk” when every square of the ship has been hit. Thus, it takes four hits (in four different places) to sink a battleship, three to sink a cruiser, two for a destroyer, and one for a submarine. The object is to sink the fleet with as few shots as possible; the best possible score would be 20. (Low scores are better.) When all ships have been sunk, the program prints out a message that the game is over, and tells how many shots were required.

Details of implementation

In this homework

Your program should have the following classes (no extra classes are needed)

The classes

- class BattleshipGame

This is the “main” class, containing the main method and an instance variable of type **Ocean**.

- class Ocean – This contains a 10x10 array of Ships, representing the ”ocean,” and some methods to manipulate it.
- class Ship – This describes characteristics common to all the ships. It has subclasses:
- class Battleship extends Ship – Describes a ship of length 4.
- class Cruiser extends Ship – Describes a ship of length 3.
- class Destroyer extends Ship – Describes a ship of length 2.
- class Submarine extends Ship – Describes a ship of length 1.
- class EmptySea extends Ship – Describes a part of the ocean that doesn’t have a ship in it. (It seems silly to have the lack of a ship be a type of ship, but this is a trick that simplifies a lot of things. This way, every location in the ocean contains a ”ship” of some kind.)

class BattleshipGame

The BattleshipGame class is the ”main” class–that is, it contains a main method. In this class you will set up the game; accept ”shots” from the user; display the results; print final scores; and ask the user if they want to play again. We also want to keep track of the lowest number All input/output is done here (although some of it is done by calling a print() method in the Ocean class.) All computation will be done in the Ocean class and the various Ship classes.

To aid the user, row numbers should be displayed along the left edge of the array, and column numbers should be displayed along the top. Numbers should be 0 to 9, not 1 to 10. The top left corner square should be 0, 0. Use different characters to indicate locations that contain a hit, locations that contain a miss, and locations that have never been fired upon.

Use methods. Don’t cram everything into one or two methods, but try to divide up the work into sensible parts with reasonable names.

class Ship

Ship will be an abstract class. We never really want to create a Ship. We always want to create a specific type of Ship.

So this means the declaration will be

public abstract class Ship

Since we don’t really care which end of a ship is the bow and which the stern, we will consider all ships to be facing up or left. Other parts of the ship are in higher-numbered rows or columns.

You don't need to write a constructor for this class–Java will automatically supply one for you (with no arguments).

Instance variables for this class

- int bowRow – the row (0 to 9) which contains the bow (front) of the ship.
- int bowColumn – the column (0 to 9) which contains the bow (front) of the ship.
- int length – the number of squares occupied by the ship. An "empty sea" location has length 1.
- boolean horizontal – true if the ship occupies a single row, false otherwise.
- boolean [] hit = new boolean[4]; – an array of booleans telling whether that part of the ship has been hit. Only battleships use all four locations; cruisers use the first three; destroyers 2; submarines 1; and "empty sea" either one or none.
- int getLength() – Returns the length of this particular ship. This method exists only to be overridden, so it doesn't much matter what it returns; an abstract "ship" doesn't have a fixed length.
- **Getters:**

  int getBowRow() – Returns bowRow

  int getBowColumn() – Returns bowColumn

  boolean isHorizontal() – Returns horizontal
- **Setters:**

  void setBowRow(int row) – Sets the value of bowRow

  void setBowColumn(int column) – Sets the value of bowColumn

  void setHorizontal(boolean horizontal) – Sets the value of the instance variable horizontal.
- abstract String getShipType()

  This is an abstract method, so obviously it has no body.
- boolean okToPlaceShipAt(int row, int column, boolean horizontal, Ocean ocean)

  Returns true if it is okay to put a ship of this length with its bow in this location, with the given orientation, and returns false otherwise. The ship must not overlap another ship, or touch another ship (vertically, horizontally, or diagonally), and it must not "stick out" beyond the array. Does not actually change either the ship or the Ocean, just says whether it is legal to do so.

- void placeShipAt(int row, int column, boolean horizontal, Ocean ocean)

  ”Puts” the ship in the ocean. This involves giving values to the bowRow, bowColumn, and horizontal instance variables in the ship, and it also involves putting a reference to the ship in each of 1 or more locations (up to 4) in the ships array in the Ocean object. (Note: This will be as many as four identical references; you can’t refer to a ”part” of a ship, only to the whole ship.)

- boolean shootAt(int row, int column)

  If a part of the ship occupies the given row and column, and the ship hasn’t been sunk, mark that part of the ship as ”hit” (in the hit array, 0 indicates the bow) and return true, otherwise return false.

- boolean isSunk()

  Return true if every part of the ship has been hit, false otherwise.

- @Override

  public String toString()

  Returns a single-character String to use in the Ocean’s print method (see below).

  This method should return ”x” if the ship has been sunk, ”S” if it has not been sunk. This method can be used to print out locations in the ocean that have been shot at; it should not be used to print locations that have not been shot at.

  Since toString behaves exactly the same for all ship types, it can be moved into the Ship class.

Extending classes

Use the Ship class as a parent class for every single ship type. Make the following. Keep each class in a separate file.

1. class Battleship extends Ship
2. class Cruiser extends Ship
3. class Destroyer extends Ship
4. class Submarine extends Ship

Each of these classes has a constructor, the purpose of which is to set the inherited length variable to the correct value, and to initialize the hit array.

Aside from the constructor you have to override this method

@Override
String getShipType()
Returns one of the strings ”battleship”, ”cruiser”, ”destroyer”, or ”submarine”, as appropriate.

class EmptySea extends Ship

You may wonder why ”EmptySea” is a type of Ship. The answer is that the Ocean contains a Ship array, every location of which is, or can be, a reference to some Ship. If a particular location is empty, the obvious thing to do is to put a null in that location. But this obvious approach has the problem that, every time we look at some location in the array, we have to check if it is null. By putting a non-null value in empty locations, denoting the absence of a ship, we can save all that null checking.

Methods for EmptySea

- EmptySea() This constructor sets the inherited length variable to 1.
- @Override

  boolean shootAt(int row, int column)

  This method overrides shootAt(int row, int column) that is inherited from Ship, and always returns false to indicate that nothing was hit.
- @Override

  boolean isSunk()

  This method overrides isSunk() that is inherited from Ship, and always returns false to indicate that you didn't sink anything.
- @Override

  public String toString()

  Returns a single-character String to use in the Ocean's print method (see below).
- @Override

  String getShipType()

  This method just returns the string ”empty”

class Ocean

Instance variables

- Ship[][] ships = new Ship[10][10] – Used to quickly determine which ship is in any given location.

- int shotsFired – The total number of shots fired by the user.
- int hitCount – The number of times a shot hit a ship. If the user shoots the same part of a ship more than once, every hit is counted, even though the additional "hits" don't do the user any good.
- int shipsSunk – The number of ships sunk (10 ships in all).

Methods

- Ocean()

  The constructor. Creates an "empty" ocean (fills the ships array with EmptySeas). Also initializes any game variables, such as how many shots have been fired.

- void placeAllShipsRandomly()

  Place all ten ships randomly on the (initially empty) ocean. Place larger ships before smaller ones, or you may end up with no legal place to put a large ship. You will want to use the Random class in the java.util package, so look that up in the Java API.

- boolean isOccupied(int row, int column)

  Returns true if the given location contains a ship, false if it does not.

- boolean shootAt(int row, int column)

  Returns true if the given location contains a "real" ship, still afloat, (not an EmptySea), false if it does not. In addition, this method updates the number of shots that have been fired, and the number of hits. Note: If a location contains a "real" ship, shootAt should return true every time the user shoots at that same location. Once a ship has been "sunk", additional shots at its location should return false.

- int getShotsFired()

  Returns the number of shots fired (in this game).

- int getHitCount()

  Returns the number of hits recorded (in this game). All hits are counted, not just the first time a given square is hit.

- int getShipsSunk()

  Returns the number of ships sunk (in this game).

- boolean isGameOver()

  Returns true if all ships have been sunk, otherwise false.

- Ship[][] getShipArray()

  Returns the 10x10 array of ships. The methods in the Ship class that take an Ocean parameter really need to be able to look at the contents of this array; the placeShipAt method even needs to modify it. While it is undesirable to allow methods in one class to directly access instance variables in another class, sometimes there is just no good alternative.

- void print()

  Prints the ocean. To aid the user, row numbers should be displayed along the left edge of the array, and column numbers should be displayed along the top. Numbers should be 0 to 9, not 1 to 10.

  The top left corner square should be 0, 0.

  Use 'S' to indicate a location that you have fired upon and hit a (real) ship,

  '-' to indicate a location that you have fired upon and found nothing there,

  'x' to indicate a location containing a sunken ship,

  and '.' (a period) to indicate a location that you have never fired upon.

This is the only method in the Ocean class that does any input/output, and it is never called from within the Ocean class (except possibly during debugging), only from the BattleshipGame class.

You are welcome to write additional methods of your own. Additional methods should either be tested (if you think they have some usefulness outside this class), or private (if they don't).

Javadocs and unit testing

Please write Javadocs for every method.

For unit testing, the usual rules apply.

There are some methods which cannot be unit tested. A method that takes in user input. Similarly a method that prints to the console (and does just that) cannot be unit tested.

Evaluation

The TAs will grade you out of 40. Note that we will weight all HWs equally in this course, so it is not the case that this one gets the most weightage or anything like that.

- Style points (5 pts total) - the usual rules about javadocs, variable naming etc still apply.

- Game play (10 pts total) - This comes down to whether or not a TA can play your game. The interface should be clear. If you have made some potentially unusual design choice, please make sure that you point that out very clearly. If you do not know what this means, it might be worth asking on piazza/office hours. If you followed all the instructions to the letter, you are fine
- Unit testing (10 pts total) - Please make sure you pass your own unit tests.
- Passing our own unit tests (5 pts total)
- Code writing(10 pts) - Make sure you correctly understand what abstract classes do. Ensure that you use the **instanceof** operation as few times as possible. There are some cases where it will be unavoidable.

  Also since there is no way for us to test the placeShipsRandomly method, the TAs will read this part of your code and make sure that you are actually doing this correctly.

Part 2

Create a class called RecursionQuestion with these 3 static methods. Remember that they all have to be recursive.

We do not need you to submit any unit tests for this question. You, however, should probably be writing them.

1. Write a recursive function killCommas that has the following method signature

   String killCommas(String s)

   that takes in a string s that may or may not have commas and it then returns a String with those commas removed.

   killCommas("a,apple,c ,d") will return "aapplec d"s

2. Write a recursive function sumDigits with the following method signature

   int sumDigits(int num)

   that sums the digits of a positive integer in a recursive number.

3. Write a recursive function that takes in an array of distinct integers and prints out all possible subsets of those integers (each subset separated by a newline). You are free to format your output in any manner you want as long as it is clear what the distinct subsets are.

   Your code must follow the following signature.

   void powerSet(int[] arr)