CIT 590 Spring 2018
Homework 8

Due 04/05/2018, 11:59:00pm

Note: this homework is more detailed than previous assignments, so start as early as you can on it.

Introduction

This homework deals with the following topics:

- Abstract classes (link)
- 2-dimensional arrays

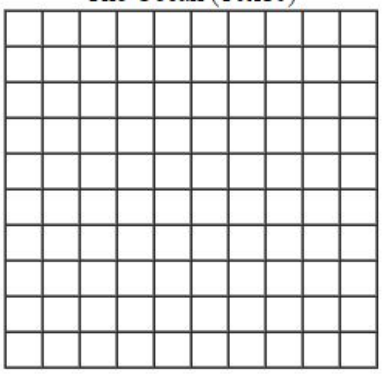
We are going to show you how to build a *simple* (only because there is no *graphical user interface* - GUI) version of the classic game Battleship.

Battleship is usually a **two-player game**, where each player has a fleet of ships and an ocean (hidden from the other player), and tries to be the first to sink the other player's fleet.

We will be doing just a **one-player vs. computer version**, where the computer places the ships, and the human attempts to sink them.

We'll play this game on a 10x10 "ocean" and will be using the following ships ("the fleet"):

The Ocean (10x10)



The Fleet

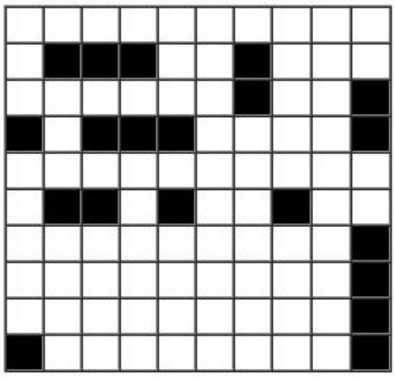
One battleship	
Two cruisers	
Three destroyers	
Four submarines	

How to play Battleship

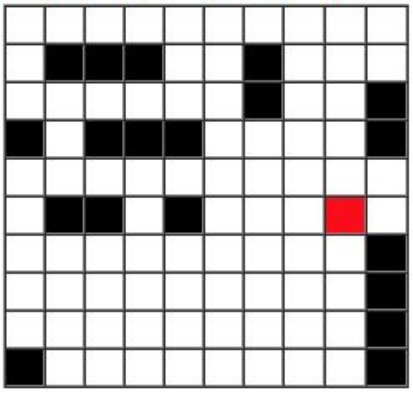
Please take a look at these rules *even if* you have played Battleship before in your life. Remember this is a **Human vs. Computer** version.

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:

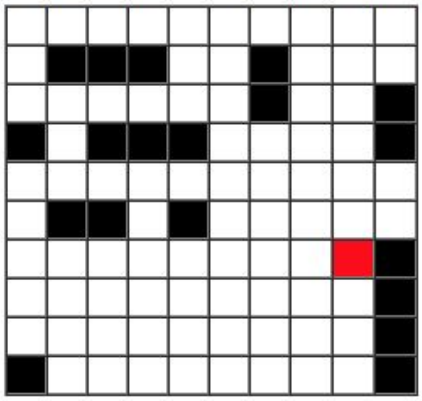
Legal arrangement



Illegal--ships diagonally adjacent



Illegal--ships horizontally adjacent



The human player does not know where the ships are. The initial display of the ocean to be printed to the console therefore shows a 10 by 10 array of '-' (see the description of the `Ocean` class' `print()` method below for more information on what subsequent `Ocean` displays will look like).

The human player tries to hit the ships, by indicating a specific row and column number (r,c). 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 (lower scores are better.) When all ships have been sunk, the program prints out A). a message that the game is over, and B). tells how many shots were required.

Details of implementation

Name your **project** `Battleship`, and your **package** `battleship`.

Your program should have the following 8 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 `Ship`

The Ship class has the following instance variables:

- `int bowRow` - the row that contains the bow (front part of the ship)
- `int bowColumn` - the column that contains the bow (front part of the ship)
- `int length`
- `boolean horizontal` - a boolean that represents whether the ship is going to be placed horizontally or vertically
- `boolean[] hit` - an array of 4 booleans that indicate whether that part of the ship has been hit or not

The methods in the `Ship` class are the following:

Getters

- `public int getLength()` - returns the length
- `public int getBowRow()` - return the row corresponding to the position of the bow
- `public int getBowColumn()` - return the bow column location
- `public boolean[] getHit()` - return the hit array
- `public boolean isHorizontal` - return whether the ship is horizontal or not
- `public String getShipType()` - this method returns an empty string in the `Ship` class. Every specific type of Ship (BattleShip, Cruiser etc) has to override this method and return the corresponding ship type.

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.

Other methods

- `boolean okToPlaceShipAt(int row, int column, boolean horizontal, Ocean ocean)`
  Based on the given row, column, and orientation, returns true if it is okay to put a ship of this length with its bow in this location; 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` - it just says if 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, index 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 is placed here in the `Ship` class.

class `ShipTest`

- Test every non-private method in the Ship class. TDD (Test-Driven Design is highly recommended.)

- Also test the methods in each subclass of Ship. You can do this here or in separate test classes, as you wish.

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 he/she wants to play again. 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 various sensible methods. Don't cram everything into one or two methods, but try to divide up the work into sensible parts with reasonable names.

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 `OceanTest`

- This is a JUnit test class for `Ocean`. Test every required method for `Ocean`, including the constructor, but not including the `print()` method. If you create additional methods in the `Ocean` class, you must either make them private, **or** write tests for them. Test methods do not need comments, unless they do something non-obvious.

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 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 **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.
  - `'S'` : Use '**S**' to indicate a location that you have fired upon and hit a (real) ship,
  - `'-'` : Use '**-**' to indicate a location that you have fired upon and found nothing there,

    - `'x'` : Use '**x**' to indicate a location containing a sunken ship,
    - `'.'` : and use '**.**' (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.
- There are some methods which cannot be unit tested, such as a method that takes in user input. Similarly, a method that prints to the console (and does only that) cannot be unit tested.

Evaluation

The TAs will grade you out of 40 and scale it down to 20:

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