Homework 10 : Battleship

(Deadline as per Canvas)

For HW10, you may work as a group (no more than 2 students). **Please mention your collaborator's name at the top of each of your code files.**

This homework is more detailed than previous assignments, so start as early as you can on it. It deals with the following topics:

- Inheritance & overriding
- Access modifiers
- Abstract classes (we'll learn about these in the next lecture)
- 2-dimensional arrays

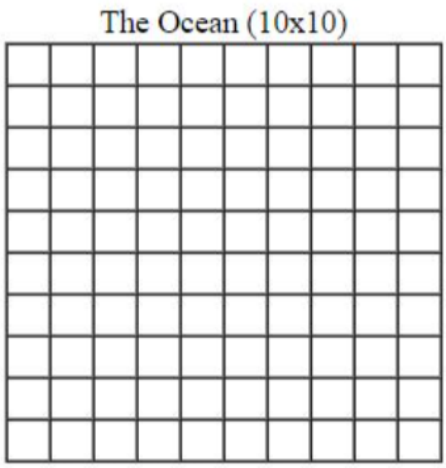
Introduction

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 Fleet

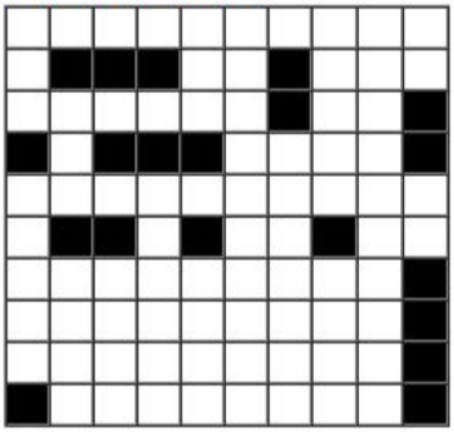
One battleship	■■■■
Two cruisers	■■■ ■■■
Three destroyers	■■ ■■ ■■
Four submarines	■ ■ ■ ■

How to Play Battleship

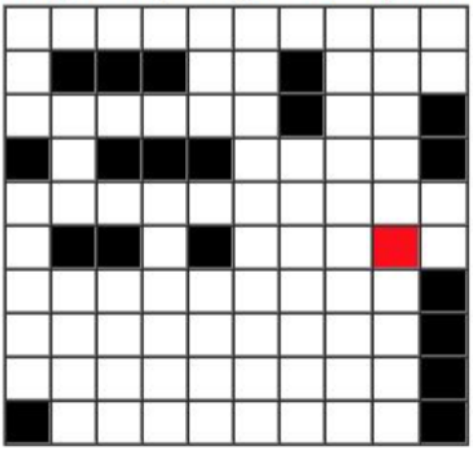
Please take a look at these rules *even if* you have played Battleship before in your life. Remember this is a **Human vs. Computer** version.

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:

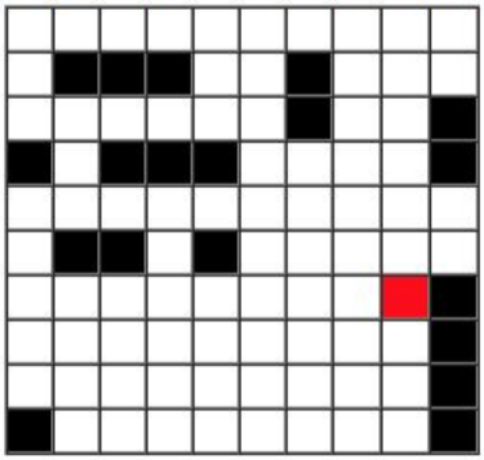
Legal arrangement



Illegal--ships diagonally adjacent



Illegal--ships horizontally adjacent



The human player does not know where the ships are. The initial display of the ocean 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 message that the game is over, and 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, which starts by creating an instance of `Ocean`
- class `Ocean`
    - This contains a 10x10 array of `Ships`, representing an "ocean", and some methods to manipulate it
- abstract class `Ship`
    - This abstract class describes the characteristics common to *all* 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.)

Abstract class `Ship`

The abstract `Ship` class has the **instance variables** below.
Note: Fields should be declared private unless there is a good reason for not doing so. This is known as "encapsulation", which is the process of making the fields in a class private and providing access to the fields via public methods (e.g. getters and setters).

- `private int bowRow`
    - The row that contains the bow (front part of the ship)
- `private int bowColumn`
    - The column that contains the bow (front part of the ship)
- `private int length`
    - The length of the ship
- `private boolean horizontal`
    - A boolean that represents whether the ship is going to be placed horizontally or

vertically

- `private boolean[] hit`
    - An array of booleans that indicate whether that part of the ship has been hit or not

The default **constructor** for the `Ship` class is:

- `public Ship(int length)`
    - This constructor **sets the length property of the particular ship** and **initializes the hit array based on that length**

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

Getters

- `public int getLength()`
    - Returns the ship length
- `public int getBowRow()`
    - Returns the row corresponding to the position of the bow
- `public int getBowColumn()`
    - Returns the bow column location
- `public boolean[] getHit()`
    - Returns the hit array
- `public boolean isHorizontal()`
    - Returns whether the ship is horizontal or not

Setters

- `public void setBowRow(int row)`
    - Sets the value of bowRow
- `public void setBowColumn(int column)`

- Sets the value of bowColumn

- `public void setHorizontal(boolean horizontal)`
  - Sets the value of the instance variable horizontal

Abstract Methods

- `public abstract String getShipType()`
  - Returns the type of ship as a String. Every specific type of Ship (e.g. `BattleShip`, `Cruiser`, etc.) has to override and implement this method and return the corresponding ship type.

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.)
  - For placement consistency (although it doesn’t really affect how you play the game), let’s agree that **horizontal ships face East** (bow at right end) and **vertical ships face South** (bow at bottom end).
    - This means, if you place a horizontal battleship at location (9, 8) in the ocean, the bow is at location (9, 8) and the rest of the ship occupies locations: (9, 7), (9, 6), (9, 5).

    - If you place a vertical cruiser at location (4, 0) in the ocean, the bow is at location (4, 0) and the rest of the ship occupies locations: (3, 0), (2, 0).

- `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. This method should return "s" if the ship has been sunk and "x" 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`

This is the JUnit test class for the `Ship` class. The *ShipTest.java* file we have provided **contains SOME of the unit tests for this assignment**. There will be more tests for the `Ship` class that we will be running your program against. Import *ShipTest.java* into your Java project and implement enough code in the methods in your program to pass all tests.

Generate additional scenarios and add test cases to each test method. You should have a total of *at least 3 distinct scenarios and valid test cases* per method (including the ones provided). For example, one scenario could be, you create and place a battleship in a particular location in the ocean by calling its *placeShipAt* method. Then you create a destroyer and see if it can be placed in a particular location by calling its *okToPlaceShipAt* method. If so, you place it. Then you create a cruiser and see if it can be placed in a particular location by calling its *okToPlaceShipAt* method. If so, you place it.

Test every non-private method in the `Ship` class. (Unfortunately, you can't test methods that are private because they are inaccessible outside of the class.)

Also test the methods in each subclass of `Ship`. You can test the `Ship` method and its subclasses in the same file.
  ```

You must include comments to explain your different testing scenarios.

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; and print final scores. 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.

For example, here’s a display of the ocean when the program first launches, and no shots have been fired.

```
  0 1 2 3 4 5 6 7 8 9
0 . . . . . . . . . .
1 . . . . . . . . . .
2 . . . . . . . . . .
3 . . . . . . . . . .
4 . . . . . . . . . .
5 . . . . . . . . . .
6 . . . . . . . . . .
7 . . . . . . . . . .
8 . . . . . . . . . .
9 . . . . . . . . . .
Enter row,column:
```

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 abstract class `Ship`

Use the abstract `Ship` class as a parent class for every single ship type. Create the following classes and keep each class in a separate file.

- class `Battleship` extends `Ship`
- class `Cruiser` extends `Ship`
- class `Destroyer` extends `Ship`
- class `Submarine` extends `Ship`

Each of these classes has a **zero-argument public constructor**, the purpose of which is to **set the length variable to the correct value**. From each constructor, call the constructor in the super class with the appropriate hard-coded length value for each ship. Note: You can store the hard-coded int values in *static final* variables.

Aside from the constructor you have to override this method:

- ```
  @Override
  public String getShipType()
  ```
  - Returns one of the strings "`battleship`", "`cruiser`", "`destroyer`", or "`submarine`", as appropriate. Again, these types of hard-coded string values are good candidates for *static final* variables.
  - This method can be useful for identifying what type of ship you are dealing with, at any given point in time, and eliminates the need to use `instanceof`

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'd 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.

The constructor for the `EmptySea` class is:

- `public EmptySea()`
  - This zero-argument constructor sets the length variable to 1 by calling the constructor in the super class
  ```

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

- `@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 the single-character "-" String to use in the `Ocean`'s print method. (Note, this is the character to be displayed if a shot has been fired, but nothing has been hit.)
- `@Override`
  `public String getShipType()`
    - This method just returns the string "`empty`"

Class `Ocean`

Instance variables

- `private Ship[][]ships = new Ship[10][10]`
    - Used to quickly determine which ship is in any given location
- `private int shotsFired`
    - The total number of shots fired by the user
- `private 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.
- `private int shipsSunk`

- The number of ships sunk (10 ships in all)

Constructor

- `public Ocean()`
  - Creates an ”empty” ocean (and fills the ships array with `EmptySea` objects). You *could* create a private helper method to do this.
  - Also initializes any game variables, such as how many shots have been fired.

Methods

- `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*.
  - To help you write the code for this method, reference the `printWithShips()` method below. It will allow you to see where ships are actually being placed in the `Ocean` while you are writing and debugging your program.
- `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 the game)
- `int getHitCount()`
  - Returns the number of hits recorded (in the game). All hits are counted, not just

the first time a given square is hit.

- `int getShipsSunk()`
    - Returns the number of ships sunk (in the 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.
    - 'x': Use 'x' to indicate a location that you have fired upon and hit a (real) ship. (reference the description of *`toString`* in the `Ship` class)
    - '-': Use '-' to indicate a location that you have fired upon and found nothing there. (reference the description of *`toString`* in the `EmptySea` class)
    - 's': Use 's' to indicate a location containing a sunken ship. (reference the description of *`toString`* in the `Ship` class)
    - '.': 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, only from the `BattleshipGame` class.

For example, here's a display of the ocean after 2 shots have missed.

```
  0 1 2 3 4 5 6 7 8 9
0 . . . . . . . . . .
1 . . . . . . . . . .
2 . . . . . . . . . .
3 . . . - - . . . . .
4 . . . . . . . . . .
5 . . . . . . . . . .
6 . . . . . . . . . .
7 . . . . . . . . . .
8 . . . . . . . . . .
9 . . . . . . . . . .
Enter row,column:
```

And here's a display of the ocean after 2 shots have missed, and 1 has hit a (real) ship.

```
  0 1 2 3 4 5 6 7 8 9
0 . . . . . . . . . .
1 . . . . . . . . . .
2 . . . . . . x . . .
3 . . . - - . . . . .
4 . . . . . . . . . .
5 . . . . . . . . . .
6 . . . . . . . . . .
7 . . . . . . . . . .
8 . . . . . . . . . .
9 . . . . . . . . . .
Enter row,column:
```

To further help you understand how to write the code for this method, here's the (high-level) logic for printing the ocean:

for each location in the 10 by 10 array (the "ocean")

- if the location contains a ship that is sunk or if the location has been shot at, and was hit or nothing was found
  - print the ship itself -- this will call `toString` in the `Ship` class or any `Ship` subclass which has `toString` defined (i.e. `EmptySea`)
- otherwise print "."

- `void printWithShips()`
  - USED FOR DEBUGGING PURPOSES ONLY.
  - Like the `print()` method, this method prints the `Ocean` with row numbers displayed along the left edge of the array, and column numbers displayed along the top. Numbers should be 0 to 9, not 1 to 10. The top left corner square

should be 0, 0.

- Unlike the `print()` method, this method **shows the location of the ships**. This method can be used during development and debugging, to see where ships are actually being placed in the `Ocean`. (The TAs may also use this method when running your program and grading.) It can be called from the `BattleshipGame` class, after creating the `Ocean` and placing ships in it.
- Be sure to comment out any call to this method before actually playing the game and before submitting your Java project.
- 'b': Use 'b' to indicate a `Battleship`.
- 'c': Use 'c' to indicate a `Cruiser`.
- 'd': Use 'd' to indicate a `Destroyer`.
- 's': Use 's' to indicate a `Submarine`.
- ' ': Use ' ' (single space) to indicate an `EmptySea`.

For example, here's a display of the ocean after creating the `Ocean`, calling `placeAllShipsRandomly`, and then calling `printWithShips`.

```
  0 1 2 3 4 5 6 7 8 9
0   d d             s
1
2 c       b b b b
3 c
4 c     s
5                 d
6 d               d   s
7 d     c c c
8
9   s
```

You are welcome to write additional methods of your own. Additional methods should have *default* access (accessible anywhere in the package) and be tested, if you think they have some usefulness outside of this class. If you don't think they have any use outside of this class, mark them as *private*.

Class `OceanTest`

This is the JUnit test class for the `Ocean` class. The *OceanTest.java* file we have provided **contains SOME of the unit tests for this assignment**. There will be more tests for the `Ocean` class that we will be running your program against. Import *OceanTest.java* into your Java project and implement enough code in the methods in your program to pass all tests.

Generate additional scenarios and add test cases to each test method. You should have a total of *at least 3 distinct scenarios and valid test cases* per method (including the ones provided). For example, a scenario could be, you create and place a submarine in a particular location in the ocean by calling its *placeShipAt* method. Then you check if a particular location in the ocean is occupied by calling its *isOccupied* method. Create and place another submarine in a particular location by calling its *placeShipAt* method. Then check if another location in the ocean is occupied by calling its *isOccupied* method.

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.

Note: Two test methods have been **completely implemented for you**, meaning, no additional test cases are required. (Of course, you're welcome to add more!). These are *testEmptyOcean* and *testPlaceAllShipsRandomly*.

testEmptyOcean tests if every location in an empty ocean is "empty". Since it's the constructor in the Ocean class that is responsible for creating an "empty" ocean (and filling the ships array with EmptySeas), the *testEmptyOcean* method can be considered the test method for the Ocean class constructor.

testPlaceAllShipsRandomly tests that the correct number of each ship type is placed in the ocean after calling *placeAllShipsRandomly*.

You must include comments to explain your different testing scenarios.

What to Submit

Please submit all the Java classes in your Java project. Make sure to include everything in your "src" folder.

If you're working as part of a team, only one student from your team needs to submit the files. Include the members of your team as part of the @author tag in the Javadocs for each class. If Brandon was working with Sarah, their Javadocs and @author tag would look something like:

```
 4
 5⊖/**
 6  * Main class for a human vs. computer version of Battleship.
 7  * Creates a single instance of Ocean. Gets user input (row and column)
 8  * for interacting with and playing against the computer.
 9  * @author Brandon Krakowsky & Sarah Broomall
10  */
11 public class BattleshipGame {
12
```

Evaluation

You will be graded out of 52 points:

- **Style** (*8 pts total*)
    - This includes, but is not limited to:
        - Adding Javadocs to every class, method, and variable, and comments to all non-trivial code.
        - Indenting properly (Cmd+i or Ctrl+i) and using { brackets } correctly in loops and conditionals
        - Naming additional variables and methods descriptively with camelCase
        - Removing unused variables and commented out code blocks/print statements used for debugging
- **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 EdDiscussion or during office hours. If you followed all the instructions to the letter, you are fine.
- **Unit testing** (*24 pts total*)
    - Please make sure you pass the provided tests as well as your own additional unit tests (*10 pts*)
    - Passing our own unit tests (*14 pts*)

  - Note, 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.
  - Note: please clearly comment on your unit tests to explain the different scenarios that you are testing. The TAs will read your tests to make sure they are distinct and valid.

- **Code writing** (*10 pts*)
  - Make sure you understand inheritance and correctly utilize overriding
  - Note: Please clearly comment on your `placeAllShipsRandomly()` method. The TAs will read this part of your code and make sure that you are actually doing it correctly.