Software and Programming 2
Coursework Four
Programming Assignment 2014-15

For submission details please see the Moodle site.

1 Purpose of this assignment

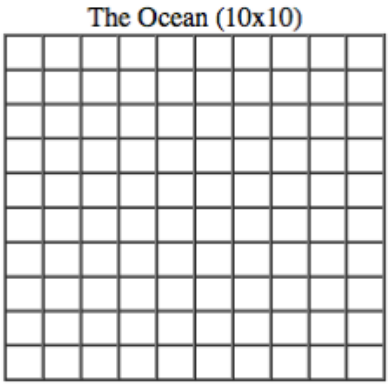
- To give you some experience with classes and inheritance
- To give you experience of writing unit tests

2 General idea of the assignment

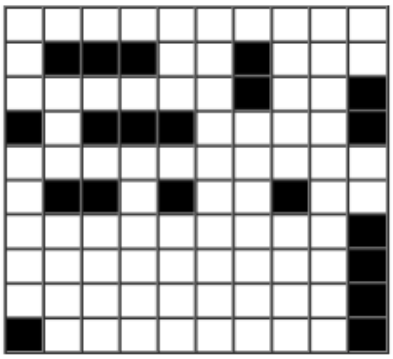
This assignment is based on a game, since games are a good source of relatively simple problems. Battleship is usually a two-player game, where each player has a fleet and an ocean (hidden from the other player), and tries to be the first to sink the other player's fleet. You will just write a solo version, where the computer places the ships, and the human attempts to sink them.

3 How to play

Initially the ocean is empty. For example,



Then 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. For example,



where the ships are represented as follows:

The Fleet

One battleship	
Two cruisers	
Three destroyers	
Four submarines	

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 — *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 redisplays 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 aim 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.

4 The classes

Your program must have the following classes (although other supporting classes are allowed):

`class BattleshipGame` This is the *main* class, containing the main method and the `Ocean`.

`class Ocean` This contains a 10x10 array of `Ship`s, 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.

Please note: It may seem silly to have the *lack* of a ship be a *type* of ship, but this little trick simplifies a lot of things. This way, every location in the ocean contains a "ship" of *some* kind.)

4.1 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 s/he 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* `1to10`. The top left corner square should be location `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 (see later).

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

4.2 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.

4.3 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:

`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.

Getters:

`int getBowRow()` Returns `bowRow`

`int getBowColumn()` Returns `bowColumn`

`boolean isHorizontal()` Returns `horizontal`

`String getShipType()` Returns the type of this ship. This method exists only to be overridden, so it doesn't much matter what it returns.

`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.

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`

Instance methods:

`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.

4.4 class Battleship extends Ship
class Cruiser extends Ship
class Destroyer extends Ship
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 initialise the `hit` array.

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

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

4.5 class EmptySea extends Ship

`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).

4.6 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.

4.7 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 `EmptySea`s). Also initialises 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. (You will probably need this method for testing. However, since it returns the *actual array of actual ships,* and could therefore be modified by some class that has no *right* to do so, use this method **only** in your unit testing.)
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 (hence my earlier restriction, now crossed out), 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 indication location containing a sunken ship, and `'.'` 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).

5 Additional requirements:

- Every method, except your test methods, should have javadoc comments. Use Eclipse's `Source` → `Generate Element Comment` to get the correct form.
- The program should be properly formatted. Use Eclipse's `Source` → `Format` to get it right.
- Every method should be short enough to see all at once on the screen.

Credits

This coursework was developed from a coursework assignment by *David Matuszek*, and incorporates features of assignments from *Cay Horstmann*, *et al* (because I can't remember).