Java Practice Test
Battleships

Thursday February 16th 2012, 15:00

TWO HOURS AND THIRTY MINUTES
(including 15 minutes reading time)

- Please make your swipe card visible on your desk.
- After the reading time log in using your username as **both** your username and password.

Credit will be awarded throughout for clarity, conciseness, *useful* commenting and appropriate use of assertions.

Important note: THREE MARKS will be deducted from solutions that do not compile. Comment out any code that does not compile before you leave.

Problem Description

Battleships is a game where two players each place a fleet of ships upon a grid, then attempt to sink their opponent's fleet by targeting specific cells on the grid.

The game is played on a 10 x 10 grid, labelled `A0` to `J9` as shown in Figure 1.[1] At the beginning of the game, each player places their fleet of ships on their own grid (which the other player cannot see). Each type of ship takes up a different number of consecutive cells on the grid, and can be placed horizontally or vertically.

Details of a player's fleet of ships are as follows:

Type	Size	Number
Aircraft carrier	5	1
Battleship	4	1
Cruiser	3	1
Patrol boat	2	2
Submarine	1	2

where **Size** indicates the number of consecutive cells taken up by this type of ship, and **Number** indicates how many of this type of ship are in the player's fleet. Figure 2 shows a Battleships grid populated with a fleet. Cells occupied by ships are shown in light grey.

Each player then, in turn, chooses a target cell (e.g. `D4`) and tells their opponent which cell they have chosen. The opponent then checks their grid. If the targeted cell is occupied by part of their fleet, they tell the opponent they scored a **hit**, otherwise a **miss**.

For example, if a player targets cell `D4` when their opponent's grid has the layout of Figure 2, this would score a **hit**, as indicated by the dark grey cell in Figure 3.

If all the consecutive cells making up an individual ship are hit, the ship is sunk. For example, if the hit to `D4` shown in Figure 3 was followed by hits to `D3` and `D5`, the cruiser occupying cells `D3`-`D5` would be sunk, as shown in Figure 4.

The winner is the player who sinks their opponent's entire fleet first.

Your task during this test is to write, in Java, a *one-sided* version of Battleships where a user plays against the computer. The computer places ships on a grid and the human player repeatedly

[1] The grid is traditionally labelled `A1` to `J10`, but this modification will simplify your code.

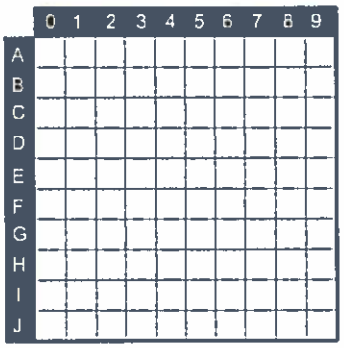

Figure 1: A blank Battleships grid.

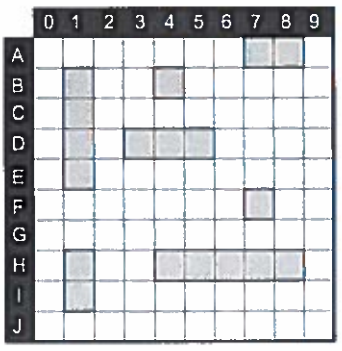

Figure 2: A Battleships grid, populated with a fleet.

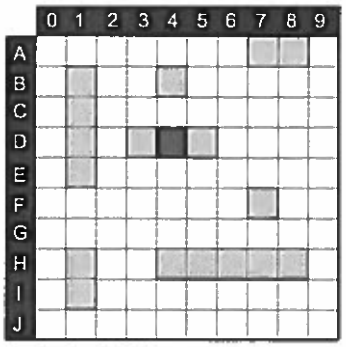

Figure 3: A Battleships grid, showing a hit in cell D4.

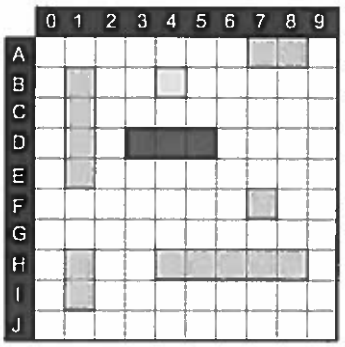

Figure 4: A Battleships grid, showing a sunk cruiser.

attacks the grid until all ships are sunk. The player's aim is to sink the computer's ships using as few attacks as possible.

Getting Started

You will find a directory `Battleships` in your Lexis home directory with a `src` sub-directory. Inside `src` is a `battleships` package directory containing 5 Java files:

- `Coordinate.java`
- `Util.java`
- `Grid.java`
- `Main.java`
- `Piece.java`

You should edit these files, filling in the stub methods as explained below. You should not change the signatures of any of the provided methods: auto-testing of your solution depends on the methods having exactly their original signatures. You may feel free to add additional methods and classes as you see fit. Any new Java files should be placed under the `src` directory.

If you work using Eclipse, you will find it already set up with a project for the source files, however the project is slightly out of sync. To sync up, do:

- Click on the `BattleShips` project and then press `F5` or click `File->Refresh`
- Click `Project->Clean->[new window]->Ok`

After this you can work on the project directly in Eclipse.

- To get you started, you are supplied with:
  1. A class `Coordinate`. An object of type `Coordinate` can be used to store an index into a two-dimensional array representing a Battleships grid.

```
 0123456789      0123456789      0123456789      0123456789
A..........     A.......##.     A.......##.     A.......##.
B..........     B.#..#.....     B.#..#.....     B.#..#.....
C..........     C.#........     C.#........     C.#........
D..........     D.#.###....     D.#.#*#....     D.#.***....
E..........     E.#........     E.#........     E.#........
F..........     F........#.     F........#.     F........#.
G..........     G..........     G..........     G..........
H..........     H.#..#####.     H.#..#####.     H.#..#####.
I..........     I.#........     I.#........     I.#........
J..........     J..........     J..........     J..........
```

Figure 5: Left-to-right: string representations of the Battleships grids shown in Figures 1–4. Water, ship and damaged ship pieces are represented by '.', '#' and '*' respectively. A miss (not shown in the figure) is represented by 'o'.

2. An enumeration `Piece` that can be used to represent the state of a square on a Battleships grid. The elements of `Piece` are: `WATER`, `MISS`, `SHIP` and `DAMAGED_SHIP`.
3. An incomplete class `Util` that defines a series of static methods, most of which you will implement as part of the test. A full implementation of one method in `Util` is provided:
   `Piece[][] deepClone(Piece[][] input)`
   This method accepts a two-dimensional array representing a Battleships grid, and returns a disjoint copy of this array.
4. An incomplete class `Grid` that defines the state and operations for a Battleships grid. The state of the grid is represented by a field `grid`, a two-dimensional array with element type `Piece`. You will implement the incomplete methods of `Grid`.
   The only methods that are initially implemented are the static method:
   `String renderGrid(Piece[][] grid)`
   which builds a String representation of a Battleships grid, and the instance method:
   `String toString()`
   which works by calling `renderGrid`. Examples of the strings constructed by `renderGrid` for the Battleships grids of Figures 1–4 are shown in Figure 5.
5. An incomplete class `Main` providing static methods:
   `Grid makeInitialGrid()`
   which returns an initial Battleships grid, and:
   `void main(String[] args)`
   which is the game's `main` method. You will implement `main` in Part 4 of the test.
   Do not change the implementation of `makeInitialGrid`. Auto-testing of your solution depends upon this method operating exactly as is.

- To read input from the user you may wish to use the class `Scanner` from package `java.util`. To create a `Scanner` that reads from `System.in`, you can write:

  ```
  Scanner input = new Scanner(System.in);
  ```

  To read a token from the scanner as declared above, you can write:

  ```
  String token = input.next();
  ```

- You may find the following instance method of the `String` class useful to determine the character at position `index` of a string:

  ```
  char charAt(int index);
  ```

- When solving each task below, feel free to define any auxiliary methods which would help to make your code more elegant.
- Your Java methods should contain assertions to check correctness conditions, including pre-conditions and post-conditions, where appropriate.

What to do

Part 1: Utility methods (6 marks).

Your first task is to implement a series of static methods, whose stubs are provided in the `Util` class.

Question 1(a): Implement `int letterToIndex(char letter)`. This method accepts a character in the range `'A'` through `'Z'`, and should return the integer corresponding to the given letter, with `0` corresponding to `'A'`, `1` to `'B'`, etc.

Question 1(b): Implement `int numberToIndex(char number)`. This method accepts a character in the range `'0'` through `'9'`, and should return the integer corresponding to the given digit, with `0` corresponding to `'0'`, `1` to `'1'`, etc.

Question 1(c): Implement `Coordinate parseCoordinate(String s)`. This method accepts a string of length 2 consisting of a capital letter followed by a decimal digit. The method should return a reference to a new `Coordinate` object whose `row` field corresponds to the capital letter and whose `column` field corresponds to the decimal digit. For example, `parseCoordinate("D4")` should lead to a `Coordinate` with `row` set to 3 and `column` set to 4.

Question 1(d): Implement `Piece hideShip(Piece piece)`. If parameter `piece` is `SHIP`, this method should return `WATER`. Otherwise the method should simply return the parameter `piece`.

Question 1(e): Implement `void hideShips(Piece[][] grid)`. This method accepts a two-dimensional array representing a Battleships grid. The method should modify the grid so that all undamaged ship pieces are replaced by water pieces, and all other pieces are left untouched.

Part 2: Initialising the grid (5 marks).

Your next task is to implement instance methods in class `Grid` that will be used to initialise a Battleships grid. A stub for each method is provided.

Question 2(a): Implement the constructor for `Grid` so that every piece on the Battleships grid is set to `WATER`.

Question 2(b): `boolean canPlace(Coordinate c, int size, boolean isDown)`. This method should return `true` if and only if it is possible for a ship of size `size` to be placed on the Battleships grid, with the top-left of the ship at coordinate `c`, without intersecting an existing ship or going beyond the bounds of the grid. If parameter `isDown` is `true`, your method should check whether the ship can be placed *vertically*. Otherwise, your method should check whether the ship can be placed *horizontally*.

Question 2(c): `void placeShip(Coordinate c, int size, boolean isDown)`. This method assumes that `canPlace(c, size, isDown)` is `true`, and actually places a ship of size `size` on the grid starting at coordinate `c`. Parameter `isDown` specifies whether the ship should be placed vertically or horizontally as in Question 2(b).

Once you have achieved this, you can test that grid initialisation is working correctly. Look at the skeleton `main` method in the `Main` class. This method initialises a Battleships grid, storing a reference to the grid in variable `grid`. If you use `System.out.println` to display the string value of `grid`, you should find that `main` displays the following output:

```
 0123456789
A.......##.
B.#..#.....
C.#........
D.#.###....
E.#........
F.......#..
G..........
H.#..#####.
I.#........
J..........
```

Part 3: Attacking the grid (4 marks).

Now you should implement instance methods in class `Grid` that control attacks on a Battleships grid. A stub for each method is provided.

Question 3(a): Implement `boolean wouldAttackSucceed(Coordinate c)`. This method should return `true` if and only if an attack to the grid at coordinate `c` would cause damage; i.e., if there is currently an undamaged ship piece at this coordinate of the grid.

Question 3(b): Implement `void attackCell(Coordinate c)`. If there is an undamaged ship piece on the grid at coordinate `c` then this should be replaced by a damaged ship piece (a *hit* has occurred). If there is a water piece on the grid at coordinate `c` then this should be replaced by a *miss* piece (a *miss* has occurred). Otherwise, the grid should be left unchanged.

Question 3(c): Implement `boolean areAllSunk()`. This method should return `true` if and only if all ship pieces on the grid are damaged.

Part 4: Implementing the game (5 marks). This part is not broken down into sub-questions.

Your final task is to write the Battleships game in the `main` method of class `Main`, using the various methods you have implemented so far. An outline for the operation of your `main` method is as follows:

- Use `makeInitialGrid` to create an initial Battleships grid
- Repeat the following, until all the computer's ships are sunk:

  - Display the player's view of the Battleships grid
  - Prompt the player to enter an attack
  - Update the Battleships grid according to whether the attack is a hit or miss
  - Print `Direct Hit!` in the case of a hit

- Tell the user how many attack attempts were required to sink all ships
- Show the final state of the grid

To show the player's view of the Battleships grid you should implement the `String toPlayerString()` method. This method should return a string representation of the battleships grid where all undamaged ships are hidden.

Note that the `makeInitialGrid` method always returns the same initial grid. In a real game this method would, of course, generate a random grid. However, in order to make it easy for you to predictably test your game, this method always generates the same initial grid.

Once again: *do not change the implementation of* `makeInitialGrid`. Auto-testing of your solution depends upon this method operating exactly as is.

Optional extra (no marks).

If you wish, you can write an additional method in `Main`:

```
Grid makeRandomGrid();
```

which places the fleet of ships randomly. Configure your program so that if `main` is invoked with the argument "random", the grid is initialised using `makeRandomGrid`, and otherwise using `makeInitialGrid`. In particular, if `main` is invoked with no arguments (as will be the case during auto-testing) then `makeInitialGrid` should be used.

Total over all parts: 20 marks.