Battleship – Sprint 1

All through this semester, you must implement five different versions of the *Battleship* game of increasing difficulty.

You must **submit one intermediate solution as latest on March 22nd at 13:00 pm including Sprint1 to 3 and the final solution on May 9th** (this date will be rescheduled within a reasonably short period, if needed). Both must be submitted, and both must be of acceptable quality, to have any chance of passing the course in the ordinary evaluation.

1 The basic game

Battleship is a popular **two-player** game played on **ruled grids**. Each player in the game receives **two grids**. One, for marking his **ships** and the other for recording his **guesses**. Players alternate turns **firing** at the other player's ships by calling out the coordinates (column and row) of one of the squares on the grid. The objective of the game is to **destroy** the opponent's fleet before they **sink** all your ships.

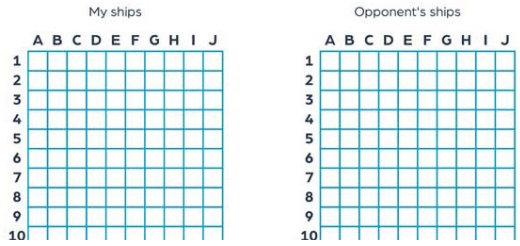


Figure 1: Typical boards to play battleship

2 Implementing the basic game

Players will be the (human) user and the computer opponent.

The game is a text-based game that uses a text-based user interface. That is, any interaction with the human user (printing boards, hit and miss messages, as well as user input) will be done through the console.

2.1 Setting up the Game

The application will record the state of the fleets using **two boards**, where different kinds of **ships** will be arranged when launching the game. All the other cells will be **water** cells.

The size of the boards as well as the number and length of ships are fixed and the same for both players. Boards are **10×10** and individual squares are identified by its column and row.

Each fleet will consist of **several ships of varying sizes**; that is, each ship occupies different number of squares in the board. In this sprint, the fleets will contain **one Battleship** (size four), **two Cruiser** (size three), **three Destroyer** (size two) and **four Submarines** (size one). Notice each one has a different size, and there may be several of each kind.

At the beginning of the game, the application will create two boards and **arrange a fleet of ships in each one at fixed coordinates**.

2.2 The Board

Boards will be **implemented as an array of integers**:

- **Numbers 1 to 4** are for each kind of ship (1 for **S**ubmarines, 2 for **D**estroyers, 3 for **C**ruisers and 4 for **B**attleships) **while they have not been shot**.
- **When a square with ship is hit**, the application replaces the content by a **negative number with the same absolute value**.
- **Water** squares will contain 0 but **when shot** will contain -10.

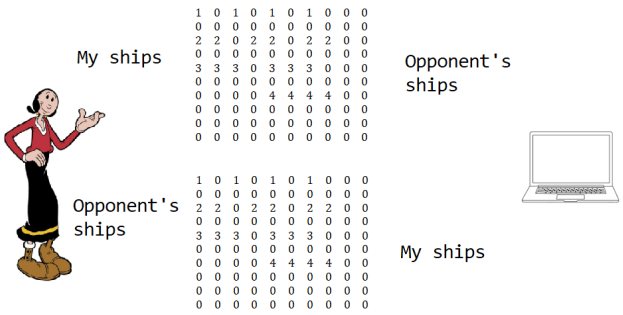
1	0	1	0	1	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0
2	2	0	2	2	0	2	2	0	0
0	0	0	0	0	0	0	0	0	0
3	3	3	0	3	3	3	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	4	4	4	4	0	0
0	0	0	0	0	0	0	0	0	0

Array 1: Initial content of the arrays

2.3 The players

Each player will hold a **name** and references to both boards. One board will be referred to as ***My ships*** and will contain the **player's ships as well as the shots by the opponent**. The other board (***Opponent's ships***) will contain the **opponent's fleet as well as the player's shots** (guesses to sink the opponent's fleet).

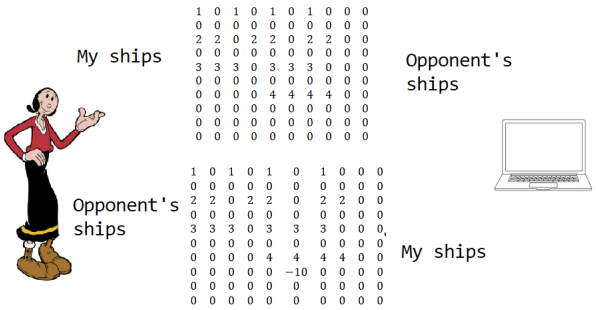
The other player will receive references to the same boards but in the opposite way. That is, the same board will be referenced to as *My ships* by one player and to *Opponent's ships* by the other player.



2.4 Playing the Game

Once initialization is done, **players take turns firing at their opponent's ships by calling out coordinates**. The application prints ***"HIT!"*** or ***"MISS"*** and changes the content of the appropriate array to **track hits or misses**.

For example, given the initial state of the board in Array 1 (image above), imagine Olivia shoots F-8, referred to column index 5, row index 7. Since the opponent (computer) does not have any ship located at this position, the application will respond with ***MISS***, and -10 must be written on this square of the array ***Opponent's ships*** (image below).



Once a player has taken their turn and the process is done, it is **time for the other player**, even if the first one hits a ship.

As soon as **all of one player's ships have been sunk**, the **game ends** and **the opponent wins** the game.

2.5 Turn processing

A player's turn consists of the **actions** below, **regardless the player is the user or the computer**. The only difference is at step 2. The **user** player always takes the **first turn**.

1. **Display boards** at the beginning of each turn. **User's board will always be printed on the left while computer's board will be printed on the right**.
2. **To make a guess**, the **player taking the turn** announces the coordinates of a target square in the opponent's board that will be shot. How to enter this guess is **the only action different for the user (standard input) and computer (randomly generated)**.
3. The application **marks the guess** made by the current player in their ***Opponent's ships*** board. Notice there is no warning when the shot refers to **repeated coordinates**. It is processed just as any other.
4. If there is a ship (or piece of it) in the target square, the application must **announce** a **hit**. Otherwise, it must inform a **miss**.
5. **The application checks whether there is a winner** (the first player to sink their opponent's ships before all their own are sunk) and prints a congratulations message if so. Otherwise, another turn is taken.

2.6 Game modes

Battleship can be configured to play in normal or debugging mode, and **the playing mode affects just the displaying, not the game procedure itself.**

By default, the game is played in normal mode so opponent's board will be displayed in a very restricted way. In debug mode, implemented just for the sake of convenience, the opponent's board is fully displayed, so the user can test the game and sink the computer's fleet quickly.

Since the game can be played in normal or debug mode, there will be a parameter to launch the game in one way or another, at the user will.

3 User interface

This section describes **the way the inner arrays must be displayed in the screen**, how the user enters coordinates, and any output produced by the application.

3.1 Print board.

Print the boards (*My ships* and *Opponent's ships*) **side-by-side** separated by two tabulators **"`\t\t`"**. Print a **title line with letters and a column with numbers for each board**. Squares will be referred by **a letter and a number.** Print "**|**" to separate columns and to separate numbers labeling rows from the content itself.

Table 1 shows the **characters to display the boards**. Notice there are different characters for each kind of ship.

Displayed square	Description
BBBB	Battleship (a **B** per square)
CCC	Cruiser (a **C** per square)
DD	Destroyer (a **D** per square)
S	Submarine
*****	Hit
Ø	Missed shot (\u00F8, empty set)
Blank space	On the user board, water squares; on the opponent's board, squares that have not been hit yet

*Table 1: Characters to **display** the content of the boards*

3.2 Display the board labelled as My ships

Use B, C, D and S to display user ships and blanks to display water squares. When one of the user ships is hit, character '*' will be displayed instead of the corresponding letter. Missed shots are displayed as '**Ø**' instead of blanks.

3.3 Display Opponent's ships in normal mode

The display of the *Opponent's ships* board depends on game mode. When playing in **normal mode**, only squares that have been shot are displayed: if they contained a ship, use the character '*' while missed shots will be displayed using the empty set symbol (see Table 1). Squares not shot yet will be displayed as blanks.

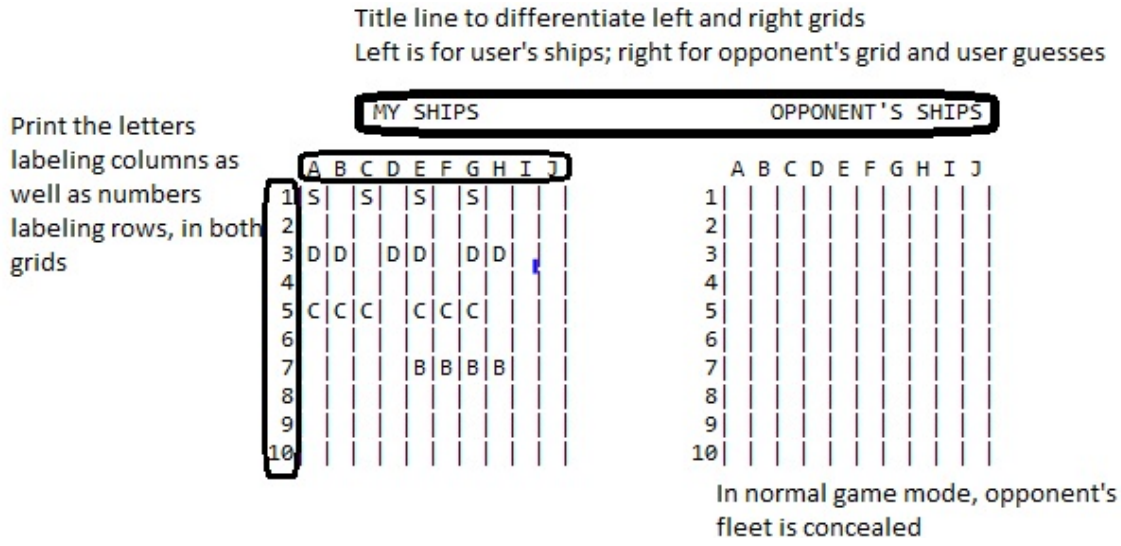


Figure 2: Initial screen of the game in normal mode

3.4 Display Opponent's ships in debug mode

If the game mode is set to **debugging**, then the coordinates of the ships on the *Opponent's ships* board will also be displayed, in the very same way as user's own ships.

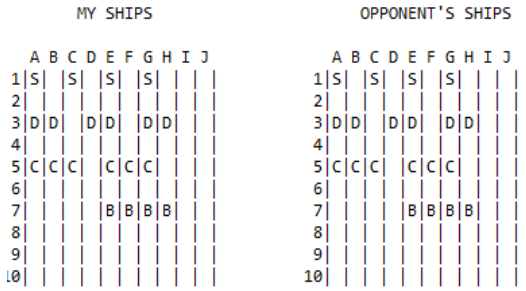


*Figure 3: Initial screen in **debugging** mode*

At the beginning of each turn, both player's and opponent's ships will be displayed. So, after some turns, at the beginning of Olivia's turn, this could be a sample output:

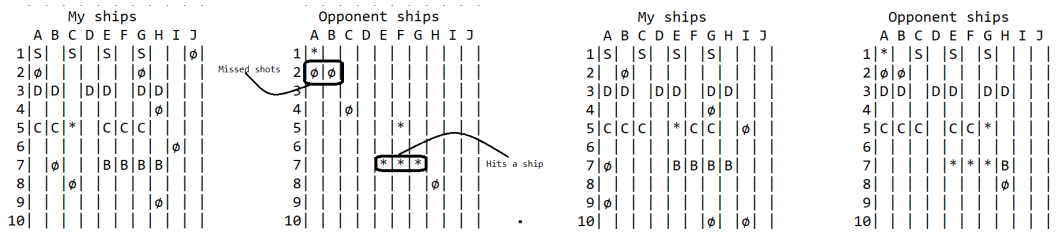


*Figure 4: Sample output after some shots in **normal** mode (left) and in **debugging** mode (right)*

3.5 User input

When the user is asked to enter coordinates, they will refer to the target row and column by using the letters and numbers displayed.

Notice there is a direct correspondence between the letters displayed and the column indexes of the array, as well as numbers and row indexes of the array. Thus, when user types A, it refers to column index 0 of the array, while number 1 corresponds with row index 0.

3.6 Manage user interaction

Apart from displaying **user boards and asking the user for typing coordinates to shoot**, there are several messages to manage the interaction with the user.

- Print a message displaying the name of the player taking the turn.
- When it is **user turn**, ask them for the next guess. Ask for column (letter) first; then row (number). When it is **computer turn**, coordinates are randomly generated.
- Print coordinates.
- Print messages with the coordinates and finally **HIT!** or **MISS** (depending on the result of the shot).

Please, try to stick to the following pictures.

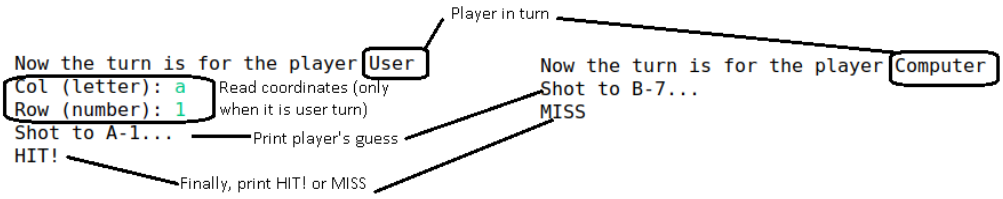


Figure 5: User interaction

The application behaves the same when it is computer turn except that it generates an **automatic guess** (**previous guesses are never repeated**) instead of reading the input from keyboard.

4 To do

4.1 UML

After reading the previous sections a couple of times and before implementing, draw a UML class diagram that models the relationships between the classes, following classes of the project: `Main`, `Game`, `Board`, `HumanPlayer`, `ComputerPlayer` and `Coordinate`. Including the rest of the classes is not required. For each class, also show attributes and public methods (do not include those attributes involved in relationships).

Try to draw the diagram without crossing lines, when possible. Lines can be either horizontal or vertical or combine both with 90-degree bends.

Do not draw it by hand, use VisualParadigm instead.

4.2 Initial project provided.

Together with this document, a minimum Java project is provided. It contains the Main class with a *main()* method to launch the game.

Rename the project as **surname1_surname2_name_battleship_sprint1** and implement the following classes **as described below**. What follows is the description of **only** the **public methods** for each class, but you may (and must) use **as many private or protected methods and private fields as needed**.

4.3 Run the project.

The `main` method gets an array of strings as argument, these are the arguments you may pass to your program. In this first sprint, only the **debug mode** argument. If the value of the argument is the string ***debug*** in uppercase, lowercase or any combination, the game will be launched in **debug mode**. Otherwise, in **normal mode**.

4.4 Class Game

It runs the main loop of the game (turns) and it is the only class in the application where output is displayed (by using the class ConsoleWriter).

Public methods:

`Constructor Game (HumanPlayer human, ComputerPlayer computer)`

It receives a HumanPlayer as first argument and a ComputerPlayer, as the second. Other game parameters as size of the boards take **default values**. That is, boards to play are **10×10**. Game mode is set to **normal** mode. It also creates a turn selector and the boards to play and passes to the players the board references.

`void setDebugMode (boolean debugMode)`

It sets the game mode. If `mode` is **false**, then game mode is set to normal (computer's fleet is concealed). If it is **true**, game mode is set to debug so computer's fleet will be displayed.

`void play ()`

It is the **only responsible for user interaction** as well as for managing the **main loop**. In this loop, players alternate **turns to shoot while there is no winner**.

Regardless the player in turn, the application behaves the same, **while there is no winner**:

- Display boards.
- Print a message with the user in turn.
- New coordinates are typed from standard input in user turn or are automatically generated in computer turn and printed.
- A shot is recorded on those coordinates in player's *Opponent's ships* board.
- A message is printed **displaying the guess** made by the current player in their Opponent's ships board. A message is printed with the result of the shot ("**HIT!**" if there is an opponent's ship in these coordinates, or "MISS" when there is no ship there).
- Check if there is a winner and print a congratulations message.

4.5 Class Board

This class represents a squared board where application will place the ships. It stores location of the ships by containing a reference to one 2D array (matrix) of integers. The content of each cell gives information about the state of the game at this coordinate as described in section 2.2.

Public methods:

`Constructor Board ( )`

It uses BoardBuilder (described later) to build a squared 2D array as Array 1.

`int getSize()`

Returns the number or rows and columns in the 2D squared array.

`boolean shootAt (Coordinate coordinates)`

It records a shot at these coordinates. If **there is a ship** there, in any state, returns true. If the user shoots the same position twice, the application will always return true, and the content of that square will not change from the first to the second shot. If there is no ship there (water), it returns false and changes the content to -10. If the user shoots the same water cell twice, the application will always return false, and the content of that square will not change from the first to the second shot.

`boolean isFleetSunk()`

Checks whether all the ships in the fleet have been sunk, returning true if so; otherwise, returns false.

`char[][] getFullStatus()`

eturns a **2D array of characters** representing the state of the board. A character at (x, y) coordinates represents the state of this square, as described in table 1 (section 3.1).

`char[][] getMinimalStatus()`

As the one before, it returns a 2D array of characters. However, it just returns the actual value of those squares in the board that **have already been shot**. For those not guessed, it returns a blank character, regardless of whether they contain a ship or not.

`int[][] getInnerArray ()`

Package protected. It returns a **copy** of the inner array of integers. Array type has a public method `clone()` that returns a duplicate copy of the same array. Useful for testing.

4.6 Class HumanPlayer

It is the class to store the human player state of the game. It contains references to two Board objects (*My ships* and *Opponent's ships*).

Public methods:

`Constructor HumanPlayer(String name)`

Creates a new object and records the name of the player. This name must always be a not null, not empty string; otherwise, we will name it "user".

`String getName()`

Returns the name of the player.

`void setMyShips(Board board)`

It sets my ships field to the argument.

`void setOpponentShips(Board board)`

It sets opponent's ships field to the argument.

`Board getMyShips()`

Returns the board recording my ships.

`Board getOpponentShips()`

Returns the board object recording opponent's fleet and my guesses.

`boolean hasWon()`

Return true if the opponent's fleet has been sunk; false, otherwise.

`boolean shootAt (Coordinate Coordinate)`

It shoots on the opponent's board. Returns true if the shot hits a ship and false otherwise.

`Coordinate makeChoice()`

Reads coordinates from keyboard and returns a Coordinate object. To that end, class ConsoleReader contains a useful method. It is the only class where user input is typed. Now, you **may assume** that the user will never type coordinates out of the board.

4.7 Class ComputerPlayer

It is almost the same as the one before. The only differences are in the constructor and `makeChoice` method.

`Constructor ComputerPlayer(String name)`

Creates a new object and records the name of the player. If `name` is null or the empty string, we will name it "`Computer`".

`Coordinate makeChoice()`

Randomly selects a coordinate from a list of not yet shot coordinates.

4.8 Class TurnSelector

It handles the turns to play. Turns always alternate, even if a player hits or even sunk an opponent's ship.

Package protected methods:

`Constructor TurnSelector ()`

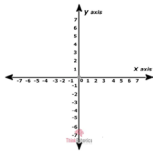
Creates a new object to alternate game's turns between two players.

`int next ()`

Returns alternating 1 and 0. Each number represents a different player's turn: 1 is for user, 0 is for computer. First turn is always a user turn.

4.9 Class Coordinate

This class represents a location in the coordinate space (2D), specified in integer precision.



Public methods:

`Constructor Coordinate (int column, int row)`

It creates a new Coordinate object referring to column *column* and row *row*.

`int getCol()`

It returns the column value.

`int getRow()`

It returns the row value.

`String toString()`

It overrides the `toString` method and returns the coordinate's values in the array: `Coordinate [ x = column, y = row ]`

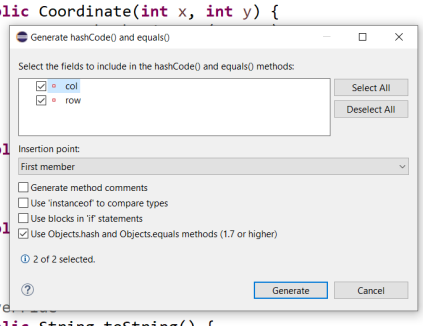
`String toUserString()`

It returns the coordinate's values in a user-friendly format. Instead of `Coordinate [x = 2, y = 0]` this method returns: **`C-1`**

`boolean equals(Object obj) and int hashCode()`

It overrides the `equals` and `hashCode` methods. `equals` return true if the argument is a `Coordinate` with the same values for x and y than this object. It is highly recommended to use the Eclipse to generate these functions (`Source → Generate hashCode and equals …`). `equals` method could be necessary in ComputerPlayer::makeChoice to avoid repeated coordinates.

```
public class Coordinate {

    private int col;
    private int row;

    public Coordinate(int x, int y) {

    }

    publ

    }

    publ

    }

    @Ove
```



4.10 Class BoardBuilder

This class **builds a squared 2D array of integers and fills it by placing numbers from 0 to 4 at predefined positions**. The size of this array is determined by the size in the Board object. Fully implemented in this first version.

4.11 Class ConsoleWriter

This class contains methods to display the game status properly (**boards are always printed from the user perspective**), to inform about the winner, etc.

Public methods:

`public void showGameStatus(Board left, Board right, boolean debugMode)`
It displays the boards (**user on the left, computer on the right**) according to the game mode. Try to stick to the description in section 3 as much as possible.

`public void showWinner (String winner)`
It prints a message with the name of the winner. **Expected messages are in the class**.

`public void showGameOver()`
It prints a message for game over.

`public void showTurn (String name)`
It prints a message with the name of the player taking the current turn.

`public void showShootingAt (Coordinate position)`
It prints the user-friendly version of the coordinates.

`public void showShotMessage (Boolean damage)`
It prints a message HIT! or MISS, depending on the parameter.

5 Tests

What follows is the description of JUnit tests you must write. Use cases to be tested are described below. Implement test methods preparing the context, running the corresponding method, and asserting expected results in any case.

5.1 Class Board

`Test method shootAt`
Use cases:

- After shooting at not shot coordinates containing no ship, the method will return false and the square will be marked as missed shot (-10).
- After shooting an already shot square that originally did not contain a ship, the method will return false, and the content will remain missed shot.
- After shooting a not yet shot square containing a ship, the method will return true and the target square will change to ship shot.
- Finally, after shooting a shot square that originally contained a ship, the content will remain the same and the method will return true.

`Test method isFleetSunk`
Use cases:

- When there are several ships afloat (not completely shot), returns false.
- When there is just one ship afloat, and it has just one position not shot, returns false.
- When all the positions of all the ships are shot, returns true.

To make writing tests easier, the following could be helpful:

- Array type has a public method `clone()` that returns a duplicate copy of the same array.
- `assertArrayEquals()` method checks that two object arrays are equal or not.

6 Minimal requirements for grading your assignment.

Before submitting, **be sure** your project runs with no errors. That is, you must develop a simple but working version of the game.

Apart from this minimum requirement, **take in mind all the following**:

- You must write JUnit for all the methods described in the test section and all must pass.
- Your program must not produce any kind of message or text through the standard output or standard error streams, other than those required here.
- All the classes must strictly conform to the public interfaces written in the skeleton. Do not change the method signatures! (name of the methods, number, type and order of the arguments, and data type returned)
- You must respect the **names of packages, package hierarchy and the classes belonging to each package**.
- Unless explicitly stated otherwise, Main class should not be changed.
- Unless explicitly stated otherwise, argument validation should be implemented for public methods using **methods in project util24**. **At least**, null objects and empty and blank Strings must be validated. At this stage, just throw `IllegalArgumentException` together with a proper message as

```
Null is an invalid value for the argument, Empty string is an invalid value for the argument, Blank string is an invalid value for the argument
```

7 Submission of the assignment

You do not have to submit this first sprint for grading, but it will be the basis for second and subsequent sprints.