David Dobor

Data Structures

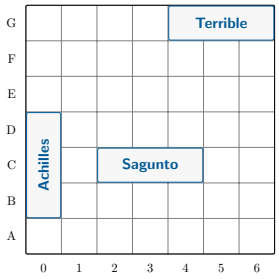
CIS 2168 Programming Assignments, Fall ’22

Contents

Assignment 1 - The Simplified Battleship Game 1
A High-Level Game Design 2
Sample Runs 2
How Do I Structure My Code? 3
How Do I Develop My Java Classes? 4
The SimpleBattleship Class 5
The TestSimpleBattleship Class 6
The SimpleBattleShipGame Class 7
The GameHelper Class 8
My Game Works! What Do I Submit? 9

Assignment 1 - The Simplified Battleship Game

In the `Battleship` game, you guess the location of an enemy ship on a 2-D grid and sink it. The goal is to sink the ships in the fewest number of guesses.

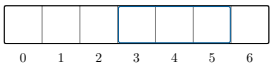


The user guesses cell coordinates like "A3," "G6," and so on. The guessed cell may be a `hit` or a `miss`. A `hit` eliminates the correctly guessed cell (a part of the ship), but it may not sink the entire ship.

Once the user guesses all the cell coordinates occupied by a ship, the ship is eliminated from the grid — this is called a `kill`.

What you'll do

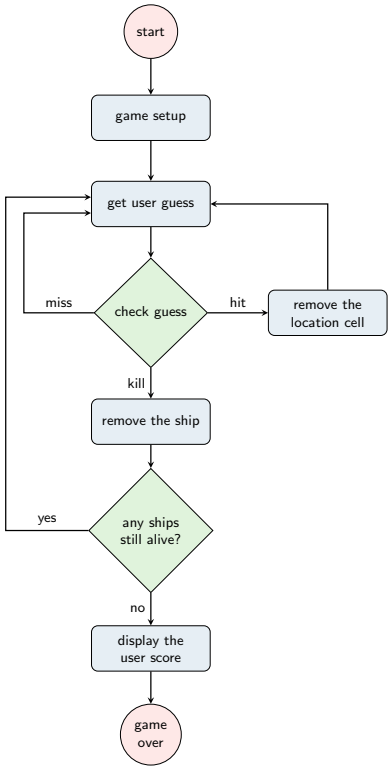
In this assignment, you'll develop a simplified version of the game called `SimpleBattleship`. Instead of a grid, the ships will be placed in a row – a 1-dimensional array. You will create only one `Ship` instance in the `main()` method of your code. The ship will occupy three consecutive cells of an array:



So instead of "A3," "G6," and so on, the cell coordinates are simply integers. For example, in the picture above, the ship coordinates are the integers 3, 4, and 5.

A High-Level Game Design

Before writing your code, you need to figure out the game's general flow. Here's how the game goes:



Game Start The user starts the game. The game setup is invoked. The setup does the following:

- creates three ships
- places the three ships onto a virtual grid

Game Play The game play begins. The play procedure repeats the following until there are no more ships left:

- prompts the user for a guess (like "A3," "G6," etc. for the 2-D game and integers for the simple version)
- checks the user guess against all ships to look for a hit, miss, or kill; if the guess is a hit, it deletes the cell; if the guess is a kill, it removes the ship from the grid

Game End The game finishes. The finish procedure

- gives the user a score based on the number of guesses

A Sample Run

A sample run of your completed 1-D version looks like this:

```
% java SimpleBattleShipGame

enter a number: 3
hit
enter a number: 2
miss
enter a number: 4
hit
enter a number: 5
kill
you took 3 guesses
```

How Do I Structure My Code?

Now that the game's logic is clear, it's almost time to begin developing your Java classes. But first, take a moment to figure out how many Java classes you'll need to build this game. Then, which of those classes should you develop first?

You'll be following sound OOP principles here (and throughout the rest of this class), which means you will *not* have one class do many different things. Instead, you'll write and coordinate several small classes, each of which does the job it is designed to do and does it well.

Here's how to organize your code. The entire game will be split into 4 classes:

Java Class	*What it does*
`SimpleBattleShip.java`	Defines what a battleship is (more on this on the next page).
`TestSimpleBattleShip.java`	Tests the `SimpleBattleShip` to ensure it works as intended. You write this one first (more on this on the next page)
`GameHelper.java`	This is a "utility" (helper) class that takes care of user input.
`SimpleBattleShipGame.java`	This is the class you run to play the game. Much of the game's logic is in the `main()` method of this class.

Since this is your first assignment, we will give the completed `GameHelper.java` and a version of the `TestSimpleBattleShip` class on the next pages (you'll write more tests of your own to make sure your `SimpleBattleShip` works).

Your job is to figure out what to put in the other two classes and how to make the whole game work. We'll simplify this for you by giving you the pseudocode for the other two classes.

What to submit? You'll be submitting the correctly working `SimpleBattleShip` and `SimpleBattleShipGame` classes only.

How Do I Develop My Java Classes?

Granted, this is not a very complicated game to develop – you do not have to build and coordinate too many classes to get this game up and running. However, we want you to follow an approach to developing your code that will save you a lot of headaches in the future when assignments get a little more involved.

Follow these steps:

- Figure out what your classes should do. For each class,
  - list the instance variables and methods
- Write *pseudocode* for the methods (this lets you focus on logic without stressing about the language syntax)
- Write test code for the methods (yes, you heard this correctly: write your test code *before* your write your classes[1])
- Implement the class
- Test the methods
- Debug, retest, and reimplement your methods as needed

Because this is your first assignment, we will provide the pseudocode for each of the classes you need. One of your main jobs in this assignment is to understand the logic of the game and translate the pseudocode into Java.

[1] Why do you think writing tests first might make sense? *Hint:* recall our class discussion. For more info, google "test driven development," or TDD for short.

The SimpleBattleship Class

The `SimpleBattleship` class has two instance variables and two methods:

Instance variable	*Description*
`int[] shipCoordinates`	Which cells does the ship occupy? (e.g.: `{3,4,5}`)
`int numOfHits`	How many cells did the user hit so far?

Method	*Description*
`String updateStatus(int guess)`	Takes an `int` for the user's guess (2, 4, etc.), checks it, and returns a result representing a `hit`, `miss`, or `kill`.
`void setPosition(int[] coords)`	Setter method that takes an `int` array (which has the three cell positions occupied by the ship – its coordinates – as `int`s).

Here's the pseudocode for the two methods. You will translate this logic into Java:

```
// receives the user guess as an integer parameter
// returns the ship status as a String ("hit", "kill", or "miss")
String updateStatus(int guess) {
   for each cell in the shipCoordinates array {
      // compare the user guess to the cell
      if the user guess matches {
         increment the number of hits
         // was this the last hit cell?
         if the number of hits is 3 return "kill" // was the last cell
         else it was not a kill, so return "hit"  // was not the last cell
      }
      else the user guess did not match, so return "miss"
    } // end for
} // end method updateStatus

void setPosition(int[] coords) {
   get the coords as an int array parameter and
   assign the coords parameter to the shipCoordinates instance variable
} // end method setPosition
```

The TestSimpleBattleship Class

Now it's time to write the *test code* for the `updateStatus` method of the `SimpleBattleship` class. Yes, we know, you are asked to write the test code *before* there's anything to test.

Look back at the pseudocode for the `updateStatus` method and ask yourself: "If the `updateStatus` method were correctly implemented, what test code would prove that the method is working correctly?" Here's what `TestSimpleBattleship` class should do:

1. Instantiate a `SimpleBattleship` object.
2. Assign a ship location (an array of 3 `int`s, like `{3, 4, 5}`).
3. Create an `int` to represent a fake user guess (3, 1, etc.)
4. Call the `updateStatus` method passing it the fake user guess.
5. Print out the result to see if it's correct (`"passed"` or `"failed"`).

Again, since this is your first assignment, we give full code:

```
public class TestSimpleBattleShip {
   public static void main(String[] args) {
      SimpleBattleShip ship = new SimpleBattleShip();
      int[] coordinates ={3, 4, 5};
      ship.setPosition(coordinates);

      int userGuess = 2; // make a fake guess
      String result = ship.updateStatus(userGuess);

      String testResult = "failed";
      if (result.equals("hit")) testResult = "passed";

      System.out.println(testResult);
   }
}
```

Note that this is just a start: you may have to make changes to this code as you develop the `SimpleBattleship` class.

The SimpleBattleShipGame Class

The `SimpleBattleShipGame.java` class is the "engine" that drives the the game. As usual, start by organizing your thoughts in psedocode. See if you can jot down the game logic before reading on. Then compare your version to ours.

Here's our version in pseudocode – everything happens in `main()`:

```
1  public static void main(String[] args) {
2    declare an int variable called numOfGuesses to hold the number of guesses
3    create a new SimpleBattleShip instance called ship
4
5    compute a random number (rand) between 0 and 4 (the starting ship coordinate)
6    make an array of 3 consecutive ints containing {rand, rand+1, rand+2}
7
8    invoke the setPosition method on the ship
9
10   declare a boolean variable named isAlive to represent the state of the game
11   set isAlive to true
12   while the ship is still alive (isAlive == true) {
13      get the user guess from the command line  // handled by a separate class
14      invoke updateStatus on the ship, passing it the user guess
15      increment the numOfGuesses
16      if the result is "kill" {
17        set isAlive to false (which means we'll break out of the while loop)
18        print the number of guesses
19      }
20   }
21 }
```

Notice that line 13 collects the user input (an integer the player enters as a guess). Substituting this line with several lines of code to handle the user input is a *bad* idea – it runs counter to good OOP principles. Instead, this operation should be handled by a separate class we call `GameHelper.java` (on page 8).

Line 14 checks the user guess to update the ship status (was the ship hit? killed?) – this operation is handled in the `updateStatus` method of the `SimpleBattleship` class which returns the ship status as a `String` (`"miss"`, `"kill"`, or `"hit"`).

Finally, the `if` statement on line 16 ends the game whenever the ship's `updateStatus` method returns `"kill"`.

The GameHelper Class

Handling user input in this version of the game is straightforward. Here it is:

```
import java.util.Scanner;

public class GameHelper {
   public int getInput(String prompt) {
      System.out.print(prompt + ": ");
      Scanner scanner = new Scanner(System.in);
      return scanner.nextInt();
   }
}
```

My Game Works! What Do I Submit?

OK, you thought through the game logic, translated your pseudocode to Java, and got the game up and running.

Great job!

But before submitting and showing your code to us, run a couple of more tests. Here are two different interactions with the game:

```
% java SimpleBattleShipGame

enter a number: 1
miss
enter a number: 2
miss
enter a number: 3
hit
enter a number: 4
hit
enter a number: 5
kill
You took 5 guesses
```

```
% java SimpleBattleShipGame

enter a number: 1
hit
enter a number: 1
hit
enter a number: 1
kill
You took 3 guesses
```

Wait! The second run (the one on the right) doesn't seem quite right: once you get a hit, simply repeating that hit two more times sinks the ship! This shouldn't be happening. Is this a bug?

Any ideas on what went wrong and how to fix it? Stay tuned – we'll address this question and more in class. It'll turn out to be an easy fix.

Here's what to do for now:

- 👉 Upload your completed `SimpleBattleShip` and `SimpleBattleShipGame` classes to Canvas before the due date.
- 👉 Demo your code to me or to your TA.
- 👉 Be prepared to discuss your ideas on how to improve your code when you demo it.