# Java Programming 2 – Lab Sheet 8

This Lab Sheet contains material based on Lectures 8.

**The deadline for Moodle submission of this lab exercise is 12:00 on Thursday 21 November 2024.**

## Aims and objectives

- Writing multi-threaded code

## Set up

1. Download and unzip **LB08_WK9_24_starter.zip file**
2. Launch Visual Studio as in previous lab (see the LB03_WK4_24 lab sheet for details)
3. In Visual Studio, select **File → Open Folder**
4. Go to the location where you unzipped **LB08_WK9_24_starter**, and highlight that folder and click on **Select Folder**
5. You should now be able to see all of your files in the explorer (if not click on **View → Explorer)**

## Submission material

Again, this lab builds on the work we have done in previous labs. As part of the starter code, you have been provided with the updated **Monster** classes from LB07_WK8_24; note that I have added a **name** field and a **getName()** method to the class.

Your task is to write classes to allow a large group of Monsters to battle together, "battle royale" style – that is, each Monster will repeatedly find another Monster to battle until it is knocked out. Each Monster will battle in its own, separate thread, and the process proceeds as follows:

- When a Monster **m1** is ready to battle, it checks the waiting area.
    - If another monster **m2** is already waiting, then **m1** will battle **m2** (see below for the details of battling).
    - If no other Monster is waiting, **m1** will go to the waiting area and wait for another monster to be ready.
- When two Monsters **m1** and **m2** battle, here is what happens:
    - First, one of the monsters is randomly chosen to be the attacker and the other will be the defender.
    - The attacker randomly chooses one of its attacks, and attacks the defender using the **Monster.attack()** method.

To make it more concrete, assume that there are three monsters **m1**, **m2**, **m3** as follows:

- M1: Fire monster, 138HP, Attacks: Attack1 (116), Attack2 (142)
- M2: Electric monster, 22HP, Attacks: Attack1 (29), Attack2 (94), Attack3 (108)
- M3: Water monster, 194HP, Attacks: Attack1 (103)

Then one possible sequence of actions is as follows:

1. M1 joins the server and waits (because there is no monster waiting to battle)
2. M2 joins the server, sees that M1 is waiting, and begins a battle
3. M1 is chosen as the attacker and chooses Attack2 to use
4. M1 attacks M2 with Attack2
5. M2 is knocked out
6. M3 joins the server and waits
7. M1 rejoins the server, sees that M3 is waiting, and begins a battle
8. M1 attacks M3 with Attack2
9. M1 rejoins the server and waits
10. M3 rejoins the server, sees that M1 is waiting, and begins a battle
11. M3 attacks M1 with Attack1
12. … M3 and M1 attack each other repeatedly until one is knocked out …

Depending on how the threads run with each other and which attacker and attack are chosen, the exact sequence may be different each time.

## What to implement

You should implement **two classes**, as follows:

- **Battle**, which keeps track of the overall battle and contains a **main** method to run it
- **MonsterRunner**, which implements the **Runnable** interface and includes the behaviour above.

Both of these classes should be in the **battle** package. Details of the two classes are given below.

### Battle

This class should have three fields: a **java.util.concurrent.locks.Lock** object which is used to control access to the battle, a **java.util.concurrent.locks.Condition** object which is used to implement the waiting behaviour, and a **java.util.concurrent.atomic.AtomicReference<Monster>** object which is used to hold a Monster that is waiting to join a battle.

The **Battle** class should also include a **main** method that runs the threads; the details of this method are given below after the description of **MonsterRunner**.

### MonsterRunner

This class should have two instance fields, a **Monster** and a **Battle**, and a constructor that sets both of those fields. It should also have an overridden **run()** method that does the following:

- While the associated **Monster** is not knocked out (i.e., hit points > 0):
    - Get the battle lock (use **lockInterruptibly**)
    - Check **waitingMonster** field
        - If the field is already set to a non-null value, start a battle with the waiting monster and then set the field to null
        - If the field is not set, set **waitingMonster** to the associated monster
    - After the battle is completed, the **run()** method should pause for 500ms (use **Thread.sleep()**)

- If the thread is interrupted at any point, you should **break** the while loop (this could happen while waiting for the Lock, while waiting for the other monster, or during the **sleep**)

When two monsters are battling, it should proceed as follows:

1. Choose which monster should attack (use **Utils.RAND.getBoolean()**)
2. Carry out the actual attack (use the static **Utils.doAttack()** method)

Please see the sample code from Friday's tutorial for an example of a **run()** method that does most of these things, except for the battling.

### Battle.main()

The **Battle** class should also have a **main()** method that creates and runs a set of threads. In summary, your method should:

- Create a new **Battle** object
- Create a list of **Monster** objects (use the provided **Utils.generateMonsters()** method)
- For each **Monster**, create a **Thread** object to run it in battle and **start()** the thread
- Wait for 10 seconds (i.e., use **Thread.sleep(10000)**)
- Interrupt all of the threads (use **Thread.interrupt()**) and wait for them all to terminate (use **Thread.join()**)

You can see the sample code from Friday's tutorial for an example of how this can work.

## Testing your code

There are no test cases provided for this lab, as testing multithreaded code is very complex. You can test your code by using different parameters in the **main** method and seeing how the behaviour changes. I strongly recommend including **System.out.println()** statements throughout your code so that you can trace the behaviour of the threads – and you should leave the **println** statements in the submitted code so that we can test it properly.

# How to submit

You should submit your work before the deadline no matter whether the programs are fully working or not. Before submission, double check that your use of variable names, comments, etc is appropriate. **Do not forget to remove any "Put your code here" or "TODO" comments!**

When you are ready to submit, go to the IOOP2 moodle site. Click on **LB08_wk9_24_submission**. Click 'Add Submission'. Open Windows Explorer and browse to the folder that contains your Java source code and drag only the *two* Java files **battle/Battle.java** and **battle/MonsterRunner.java** into the drag-and-drop area on the moodle submission page. **Your markers only want to read your java files, not your class files.** Then click the blue save changes button. Check the .java file is uploaded to the system. Then click **submit assignment** and fill in the non-plagiarism declaration. Your tutor will inspect your files and return feedback to you.

## Outline Mark Scheme

Your tutor will mark your work and return you a score in the range **A1** to **H**.

## Possible (optional) extensions

Here are some additional things to try if you want further practice with multithreaded programming:

- At the moment, the battle runs for a specified amount of time; see if you can figure out how to make it run until there is only one Monster that is not knocked out.
- Try writing the output to a log file instead of to System.out.