

**CSC2002S - 2021**

**Assignment 2 – Report**

**HMMDEN001**

This report has been divided into sections with detailed headings.

## **Introduction – Assignment brief specifics**

The purpose of this assignment is to use a simple word-typing game to investigate the implementation of multi-threading in java, while maintain thread safety and ensuring a good level of concurrency so the performance of the game is relatively smooth. Concurrency is when there is efficient and correct access between the threads to the shared resources (in this case it will be the variables shared between classes). The code also has to account for liveliness and no deadlock, which are common errors when working with multiple threads in code.

Due to the game needing multiple threads in order to be run smoothly, there will need to be a few adjustments in the skeleton code given to us to ensure that thread safety is present and maintain a good level of concurrency. These additional changes on the variables will be on the variables that are shared between classes, i.e. it is possible that more than one thread will be requiring access to it in order to read in that current value and update it. Thus all variables have to be thread local, this can be achieved with locks and/or atomic signatures; this will ensure that thread A won't read then thread B comes along and reads an incorrect value and updates it before Thread B can see the value Thread A was supposed to change it to. This will avoid race-conditions, and bad interleaving's that can cause these variables to become inaccurate; thus the game to not have a smooth flow

## **Explanation of skeleton code and code added**

### **WordDictionary.java:**

This class is pretty simple and wasn't needed to be adapted much in the purposes of this assignment. The job of this class is to correctly populate the 'word dictionary' which is used as an array of words for the word game itself. The method `getNewWord()` is synchronized to ensure that only one thread can start a new word falling down from the top of the screen, this prohibits more than word being loaded onto the screen when a word is effectively missed or caught in the duration of the game.

### **WordPanel.java:**

The `WordPanel.java` program extends `JPanel` and thus controls the GUI interface with the specific dimensions. This class contains all the information pertaining to the componenst that make up the GUI, implementing the `Runnable` interface.

The `paintComponent(Graphics g)` is how words get printed onto the screen, getting each word from `WordRecord` and using their relative x and y cords to get them onto the right position on the screen. These cords are stored in different threads to allow the words to fall at different speeds.

The `run` method is what is called upon from the `WordApp`'s main method to effectively start the animation. I used a timer that will create a thread and run the `actionListener` after a specified milliseconds (500 in my case). The thread will drop each words on the screen and then repaint it consecutively, and this will effectively result in a falling image. If that specific word has fallen off the screen, then the boolean flag in `WordRecord` (`dropped`) will become true and that's when we reset that word using the `resetWord()` method in the `WordRecord` class.

### **Popup.java:**

This is a program I created and added. The purpose of this program is to allow for a more user-friendly interface. This is from the Java Swing library. It contains a public static method that takes in the parameters of Strings that will describe what the message will say in the pop up and the title of the pop-up.

It uses the `JOptionPane` package and there's method in there called `showMessageDialog` that creates a simple pop-up message that will disappear when the button at the bottom is clicked.

This has been incorporated in the blocks of code in the `WordApp.java` program before the actions of the buttons are executed.

**WordApp.java:**

This is the class that contains the main method from where the game is run. It is responsible to set up the interface of the GUI and also updates the relative score class atomic variables when necessary.

Code was added to ensure functionality of the existing buttons – start, stop. Code was also added to add a quit and pause button, adding more button functionality to the game. The even handling was done in this class by the ActionListener. The blocks of code were also needed to be added for the ActionListener effectively ‘heard’ that specific button being pushed – whether it was the start button so that the animation of the words falling down starts (game starts) or the quit button so that the game can exit.

**WordRecord.java:**

This class is what controls the correct x and y value of the falling word on the screen. When this class is called upon the falling speed is randomized, ensuring that the words fall down on the screen at different speeds as specified in the assignment brief.

The get methods and set methods in this class have to have the added signature of being ‘synchronized’ to the methods to ensure that only one thread can use them at a time. This ensures no confusion can occur as to if multiple threads both change the coordinates of a falling word simultaneously.

**Score.java:**

The Score class contains the variables that are displayed on the screen in the duration of the game; these include the score, caught and missed counters respectively. It also contains useful get methods and set methods that were apart of the skeleton code given to us.

This class had to be adapted however, to ensure thread safety and correctness for those variables only. The operations that are done on these, now atomic variables, can’t be indivisible – it’s effectively an all or nothing approach when it comes to multiple threads updating these values, no partial updates can occur. This allows the variables to have accurate values.

The get methods and set methods are adjusted using the ‘synchronized’ locks to ensure those operations done in those methods are atomic and bad inter-leavings won’t occur. These methods include getMissed(), getCaught(), getScore() and getTotal(). Other methods such as missedWord(), caughtWord() and resetScore() are used when updating the relevant atomic variables of the class.

## **Concurrency features used in the code:**

### **WordPanel class:**

This class is already up to standards of maintaining concurrency as the Boolean flag 'done' is already made volatile in the skeleton code. It needs to be volatile to ensure that the memory is stored in main memory and not in the CPU cache (this cache can be different for all the multiple threads running), this ensures that the correct value of the Boolean variable can be obtained by the threads.

### **WordApp class:**

This class only has a Boolean flag that is made volatile in the skeleton code already, the reasoning for this is the same as seen in WordPanel class.

### **Score class:**

The variables within this class namely; the missed, caught and score counters all have been made Atomic Integers to ensure that when threads read and write to these values (essentially update) that only one thread can do this at a time. The operations on these threads become thread-local and atomic. That's why it was the correct choice.

Another feature of atomic variables is that they ensure that the memory of the variable is stored in the main memory, and not each CPU caches. This is done to maintain the correctness of the variables, no threads access the wrong values.

Aswell as the aforementioned features, synchronization is used for the methods inside this class to ensure that only one thread can use them at a time, not more than one thread can get a lock for that method at a time.

This avoids race condition errors aswell as data races.

### **WordRecord class:**

All the accessor methods are synchronized, ensuring that only one thread can obtain a lock for it and use it at a time (as mentioned before). For example with the setX() method, if more than one thread could use it at a time then it would mess with the coordinates of words on the screen.

## **Code added to ensure:**

### **Thread safety:**

This is the thread safety for both shared variables and the Swing library that's incorporated in the code. This has been ensured due to the fact that we haven't allowed more than one thread use methods through the synchronization signature of the methods. This is vital due to the fact that if more than one thread use certain methods it can cause the game logic to become inaccurate and ruin the overall 'flow' of the game.

Another point worth mentioning is to keep the data of shared variables in the main memory and not in the CPU cache. For example if two different threads are sharing one variable, and that variable isn't protected with volatile or atomic, those two variables may see different values of the same variable. This is due to the fact that threads, when storing the data in their own CPU cache, often end up with different values with each other – even when doing the same operation! Thus it's vital to ensure that variables' data are protected with the volatile or atomic keywords. This will avoid data races.

Data races occurs when multiple threads access the same memory for reading and writing purposes, and they end up 'racing' to get to that data first.

### **Thread synchronization:**

This is needed to ensure that the threads work in a way which they are intended to. It is ideal to have synchronization blocks and methods where it is possible that more than one thread may try access it at any point in time. Locks weren't used in this regard as the critical section would've been too short and it wouldn't have been sufficiently protected, thus instead the synchronization was used in order to specify the length of protection, ensuring full protection on shared resource.

Allowing for only one thread at a time to have a lock and use that method.

The synchronization on the method protects and locks the whole method whilst the synchronization on blocks of code lessens the area to be protected and locked (reduced critical section).

**Liveness:**

This aspect of the code must be maintained in good standing to ensure that the code runs smoothly and correctly. This entails that no threads are 'starved' of resources, they aren't blocked unnecessarily and that no threads are 'greedy' either, but holding certain shared resources for too long. If locks are used incorrectly and there may be a situation where a thread holding a lock on one method needs another method, but a different thread holds a lock on that one and it needs the method that the first thread is holding onto, this results in a deadlock. The threads essentially fall in this indefinite loop where they keep waiting for each other but nothing happens. However the code is structured in such a way that there isn't an instance where this circular waiting can occur, ensuring no deadlock nor thread starvation – allowing the code to run smoothly and correctly.

**No deadlock:**

Deadlock is when two threads are staring each other down, and neither can move forward nor back. They both require data from each other in order to move forward with the logic they need to perform, but neither is willing to give the other that data. This may continue indefinitely if not addressed and fixed. This can happen when one synchronous method is needing to call upon another synchronous method and a circular motion is seen to happen, and both threads become stuck waiting for the other thread to let go of that lock they have on the synchronous method. Since there is no structure of this present in the code for the word game, no deadlock can thankfully occur.

## **How to validate the program?**

There are a few ways to experiment and ensure that the code is correct and that the game is running correctly.

The most simple and easy way is to play the game with default dictionary at first and see if everything works as expected. Then I tried a large txt file with a few thousand words and played the game for as long as I could, everything worked as it should. In the duration of the game the buttons can be tested as well, I started the game, paused it, ended it and quit the game too. All of the buttons worked as intended, regardless of the speed the words were falling at. I also tested the buttons when the words were falling really fast down the screen.

Not only did I run the game with a large dictionary for the words to be generated from, I also played the game dozens and dozens of times – upwards of 48 times. The score counter as well as the other counters worked as expected. Meaning no data races occurred. The game also didn't just stop for no apparent reason either – meaning no deadlock occurred which is ideal.

We have to avoid data races and interleaving's. Data races occur when multiple threads try access the same memory location with read and write operations to do, and they end up 'racing' to get to that data first.

Data races may occur with regards to the score class and the instance variables therein. As if multiple threads try to access the score variables and update the caught, missed and score variables simultaneously – this read and write action of the threads can cause problems. This will result in a data race, and will cause the game to update the variables incorrectly



## **How does my design conform to the Model-View-Controller pattern?**

The Model-View-Controller pattern is a software design pattern that's used for interfaces, this divides the programming logic into three elements. Namely these are; the model which is the applications data structure, the view which is the interface (essentially the GUI) that the user can see and thirdly there's the controller which accepts commands from the user. When the user presses a controller, the controller then specifies which control was pushed to the model and certain data gets altered and updated to the screen which the user views.

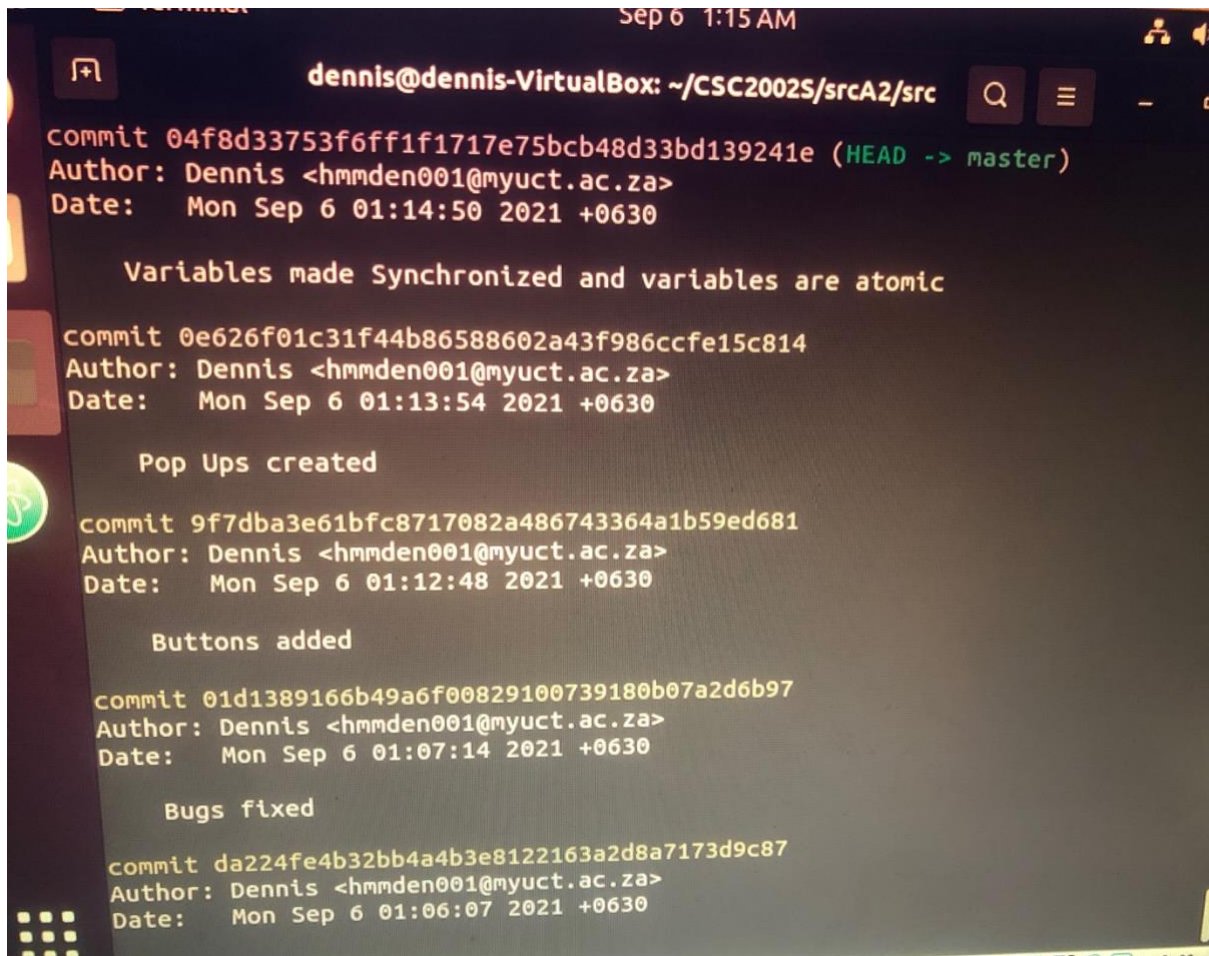
I have adhered to this pattern by adding in code in the WordApp.java class where the start, end and quit buttons are pushed respectively. These blocks of code only execute if these buttons are pushed, the ActionListener will essentially 'listen' for these buttons to be pushed then run those blocks of code. Those blocks of code will alter the model and change the data inside accordingly.

In short, the Model is made up of the code needed to run the game correctly, the View is the GUI that pops up and that the user can interact with and lastly the Controller is how the model knows when to manipulate the data correctly when the user presses buttons or essentially plays the game.

## **Additional features that were added:**

A class called popup.java has been created to allow for a more user-friendly interface. I understand that a popup was necessary at the end of gameplay, but I thought it was a good idea to also have pop-ups when the start button is pressed the words will immediately start falling and when they other buttons are pressed too. If pressed erroneously it will start falling and the user will get to a bad start to the game. So the pop-up will essentially 'pop-up' onto the screen and only when the button is pressed then the animation will start.

## **Git Log snapshot**



```
terminal
Sep 6 1:15 AM

dennis@dennis-VirtualBox: ~/CSC2002S/srcA2/src

commit 04f8d33753f6ff1f1717e75bcb48d33bd139241e (HEAD -> master)
Author: Dennis <hmden001@myuct.ac.za>
Date: Mon Sep 6 01:14:50 2021 +0630

    Variables made Synchronized and variables are atomic

commit 0e626f01c31f44b86588602a43f986ccfe15c814
Author: Dennis <hmden001@myuct.ac.za>
Date: Mon Sep 6 01:13:54 2021 +0630

    Pop Ups created

commit 9f7dba3e61bfc8717082a486743364a1b59ed681
Author: Dennis <hmden001@myuct.ac.za>
Date: Mon Sep 6 01:12:48 2021 +0630

    Buttons added

commit 01d1389166b49a6f00829100739180b07a2d6b97
Author: Dennis <hmden001@myuct.ac.za>
Date: Mon Sep 6 01:07:14 2021 +0630

    Bugs fixed

commit da224fe4b32bb4a4b3e8122163a2d8a7173d9c87
Author: Dennis <hmden001@myuct.ac.za>
Date: Mon Sep 6 01:06:07 2021 +0630
```