Assignment 2: Java Parallelism and Concurrency

AIM:

on this assignment, we are tasked with designing a multithreaded Java Program, that ensures both thread safety and sufficient concurrency for it to work well.

The program will produce words that fall from the top of the screen, and a user attempts to write every word before it touches the bottom of the screen. If a word is written accurately, the number of captured words is added then the score increases by the length of the typed word. The word then vanishes, and a different one starts coming out.

If a word is not written accurately before it touches the bottom of the screen, it fades, then the number of missed words is incremented. Another word then begins falling off the top of the screen.

The importance of the game is heavily dependent on its use in concurrence. It relies on the smooth and easy interaction between the user and the game while demonstrating the value of data integrity when distributing resources and acting concurrently on multiple threads.

Classes:

The following classes were provided to us as a "skeleton package" some of the classes have been modified for them to work.

Score:

Is used to keep count of the caught word and the missed words, it is also responsible for producing our high score and game score after a user finishes playing the game. When enabled it will display the caught words, missed words, high score, and game score.

I did not have to change a lot of things in the score class, created the high score field that allowed me to record the highest score obtained in every gameplay that the user passes, and then had to synchronize the getter and setter methods that were already in the class.

Synchronizing methods:

I had to guarantee thread safety because it is very crucial in the Score class. multiple classes will be accessing the same data, probably completing multiple operations on the data values, and the Atomic Variable method will no longer be adequate. Alternatively, the synchronized keyword on the getter and setter methods are required. The exclusion to this is the toString() method which does not require thread-safety, the reason being it is only ever called once, from a single thread, when the game is over.

WordApp:

I used wordApp as "main" its function is to initialise the classes that I created.

Most of the methods that I used to build my program stem from wordApps, originally the methods were created for wordApp but it had a few missing functions and its buttons were not functioning, so far what I have been able to do is to access the default dictionary that was given to us in order to produce the falling words, I tried to create threads and new classes in order to go with the assignment instruction of using model view controller pattern.

WordPanel:

I used WordPanel to refresh and update the panel frequently, as to reflect the words coming from the top of the screen.

WordPanel uses a runnable interface, but it contained an empty run () method, so I added the repaint() method. This allowed my WordPanel to start running but it gave me an error at the start of the code after variable declaration, so I created WordPanel(){} to go with the runnable method and created an override method.

Thread Safety:

throughout the game and throughout my coding, I was trying to guarantee that thread-safety is achieved, solely to demonstrate that the characteristics of the program guarantee no conflicts transpiring. this is seen with the class called WordRecord, the get methods and set methods for each word's coordinates, text, and falling speed were not synchronized, they were only accessed from an individual thread. In conclusion with the reasons provided I can say that thread safety wasn't required while I was adding blocks of code to each class.

Ensuring liveness and no DeadLock:

For each class that I created I had to be aware of thread-safety measures for me to produce a program that insures me to use the correct forms of concurrency. But it is only in the score class, where I had to synchronize only the set and get methods, which was one of the few classes where synchronization was used, it was used very loosely and this guaranteed me with an efficient structure of code, ensuring that deadlocks do not occur. The liveness of the program was further cemented by the fact that each thread had its own function to handle, with each method being independent of each other to provide optimal functionality.