JDGHOL001
CSC2002S ASSIGNMENT 2

**CLASSES**

WordApp

WordApp was modified to include and implement a quit button. It was also modified to give functionality to the start and end buttons. The end button was intended to refresh the pane so that a new game could start however I did not get this part working.

The start button iterated over the number of words provided as a parameter that were to fall at a given time, with each iteration instantiating a new WordPanel object and calling start. This triggers WordPanel to create n number of threads where n=number of words to fall at a given time. There is one thread per word, essentially.

In the main method, before the GUI is set-up using the method setUpGUI(), SwingUtilities.invokeLater is called. Swing is not thread safe by default, therefore, to make it thread safe updates to the GUI via Swing must occur in the Event Dispatch Thread (EDT). invokeLater() delays the GUI creation to make sure the GUI creation takes place inside the EDT. Calling invokeLater() guarantees that setUpGUI will run on the EDT.

WordDictionary, score, wordRecord

Unchanged unless there were getter or setter methods that were not synchronized, then those were synchronised.

WordPanel

The bulk of my changes and how my program runs is inside WordPanel.

**New methods/constructor**

Additional constructor:
I added another constructor (that of which threads were created) that took in as a parameter a counter value and a WordRecord object as well as a WordRecord array.

Start()
When called start created and started a thread

incrementMissed()
Incremented the missed counter which was used to track the number of words that reach the red zone before the user typed them.

getMissed()
Used to retrieve the value of the missed variable

incrementCount()
Incremented the counter that counted the number of words that had fallen

getCount()
Used to retriebe the value of the counter

**Updated methods**

paintComponent()

If the words had not yet reach the red zone, they were continuously repainted on the screen.

If they had reached the red zone an atomic counter (counting the number of words dropped) was incremented. If the total number of words allowed to fall had not yet been reached then the word was reset using resetWord() from WordRecord. The label that stated the amount missed was then updated as well (by accessing the JLable variable from WordApp and calling setText()). If the total number of words allowed to fall had been reached then resetPos() from WordRecord was call on all the words on screen (setting their Y value to 0). And done was made true, which ended the while(!done) loop in the method run().

run()

This method ran in an indefinite loop that only ended when the total number of words had fallen and done was set to true in the paintComponent() class.

In a synchronized block, the run method checked to see if the word the user typed in matched one of the falling words. If it did, then the falling word was removed from the screen and the score values were updated. The score label and the caught label on the screen were then updated using these new values.

If there was no word typed by the user or the word didn't match any of the words falling then drop() was called, incrementing the y value. The value of the word had now changed so it was updated in the WordRecord array word. The thread then slept for word.getSpeed(), a random time assigned to the word by WordRecord.

**CONCURRENCY FEATURES BY CLASS**

**Score class:** All the get and set methods are synchronised to guard against a thread updating a value while another thread is reading.

**WordPanel class:**

Atomic variables:

1. To count the number of words that had “fallen”
2. To count the number of words that had been missed

These were used so ensure thread safety of these values so that no thread could interleave and update them or read them incorrectly.

Synchronized block:

A synchronized block was used over a critical section that updated the value of score and caught and updated the “score” and “caught” labels in the GUI.

Volatile:

There is a volatile Boolean variable “done” that enables concurrency because it is visible to all threads. When it is set to false the while loop in run() stops. Being volatile it doesn’t matter which thread sets it to false, they can all see its been set to false and the while loop terminates.

**WordRecord class:** All the get and set methods are synchronised to guard against a thread updating a value while another thread is reading.

**THREAD SAFETY AND THREAD SYNCHRONIZATION**

Ensuring thread safety means ensuring that no two threads access the same shared variable at the same time. There were several areas in this application were threads needed to be prevented from accessing the same shared memory simultaneously.

In the WordPanel class, if the user correctly entered one of the words that was falling then score needed to be updated by the length of that word and caught needed to be incremented by 1. The score class is accessed and written to by many of the threads. Without synchronization, another thread could have interleaved and incorrectly updated the score. To ensure that the score was updated by the length of the word that was “caught”, the block of code needed to be synchronized.

Also in WordPanel class, the labels in the frame are being updated. The blocks of code in which that happens are synchronized to avoid more than one thread trying to update the label at once.

All getter and setter methods were made synchronous since multiple threads accessing or changing these variables at the same time would cause corruption.

As said earlier, Swing is not thread safe. Therefore, to make it thread safe updates to the GUI via Swing must occur in the Event Dispatch Thread (EDT). I used invokeLater() in the main method when running setUpGUI. This delayed the GUI creation to make sure the GUI creation took place inside the EDT i.e., it guaranteed that setUpGUI will run on the EDT.

**LIVENESS**

Liveness refers to the applications performance. Liveness ensures that all threads have access to resources and aren't "starved". The program runs at a good speed as the synchronized blocks are relatively fine-grained and therefore do not hinder the performance of the program.

**NO DEADLOCK**

Deadlock is when two or more threads are blocked because they are waiting for each other to finish (they might need to access a lock or data from the other). I did not have synchronized methods inside synchronized methods, so my design prevented deadlock. My threads were also not reliant on each other.

**HOW I VALIDATED MY PROGRAM**

I ran my program many times on two different datasets and checked to see if the results were as expected. The score, number of missed words and number of caught words was correct and therefore no race conditions occurred. The application did not stop running at any point which means that no deadlock occurred. Whilst I validated my program through experimenting, I did not do so exhaustively so there may be a probability that there will be a race condition on a certain a bug that I have not discovered it found.

I debugged and discovered race conditions mainly by using print statements. When updating the score, I would print was the score was before and after it had been updated, and what the correct update should've been. This helped me determine where bad interleaving had occurred and helped me fix it.

**HOW MY PROGRAM CONFORMS TO MODEL-VIEW CONTROLLER**

In my program, the model comprised on the classes WordDictionary, the array of WordRecords and the Score. These are the functionalities of the program.

The view is comprised of WordPanel and WordApp. WordApp sets up the GUI and retrieves the user input and WordPanel repaints the words and new values of score, missed and caught to the screen.

The controller is the threads which produce the animation, update the score etc. My threads ran in WordPanel.

An example of how the Model-View-Controller design plays out in my program:

1. When the user types a word, the word is compared to the words in the threads and if one of the words is the same the word is deleted and the Controller/threads update the model (new score), and then the model updates the view.

In this way my design conforms to the Model-View-Controller. However, I think that if I were to create something larger or more complicated, I would give the Controller/threads its own class. But for the purposes of this assignment, I didn't find it necessary.