TAFADZWA NYAZENGA
NYZTAF001

CSC2002 Assignment 4: Concurrent Programming

Introduction and Aim

The objective of this assignment was to create a typing game. The game has words falling from the top of the window and the objective for the user playing the game is to type out all the words, one at a time, before the words reach the red section at bottom of the program window and the game will count the number of words 'caught' or rather typed correctly and in time and the score is a total of the length of all the words captured. Since the words will be falling simultaneously, some concurrency was required.

Class Descriptions

Score.java

This class, which was provided, tracks the number of words caught, number of words missed and the score of the user as the game is running. It has the following methods:

- *getMissed()*: return the number of words missed
- *getCaught()*: return the number of words successfully captured
- *getTotal()*: return the total number of words that appeared during the game
- *getScore()*: return the user's score
- *missedWord()*: increase count of missed words by 1
- *caughtWord()*: increase count of caught words by 1 and the score by the length of the last captured word.
- *resetScore()*: reset all the counters to 0

No changes were made to this class from that provided.

It is part of the Model in the MVC pattern.

WordDictionary.java

This class is responsible for either providing a default dictionary of words for the game of alternatively adding a custom dictionary to the game from a file. No changes were made to this class from that provided.

It is part of the Model in the MVC pattern.

WordRecord.java

This class stores each word and is responsible for tracking its position and status. No changes were made to this class.

It is part of the Controller in the MVC pattern.

WordPanel.java

This class is responsible for panel that will contain the falling words and the associated animations. The following methods were added from those provided:

- *checkCompletion*(): check if the game is still in progress
- *completed():* set when the game is completed
- *incomplete():* used as reset to restart the game
- *run():* this method initializes all threads responsible for the animation of each word (1 per word). The word drops every 300 milliseconds and the thread terminates when the word reaches the bottom of the game window.
- *checkInput()*: this method is used to check if the word typed is one of the words on the screen.

It is part of the Controller in the MVC pattern.

WordApp.java

This class contains the program's main class. It is responsible for running the game and handling all the display components like the buttons and the text box. The following features were added:

- *closeGameB*: this is the quit button
- *runnable keepScore*: is responsible for updating the score on the screen

It is part of the View in the MVC pattern.

Concurrency Features

The WordApp is generating 3 types of threads:

- keepScore thread is responsible for continuously updating the score.
- Each word will have its own thread that is responsible for its animation.
- Main thread which will run the program

Synchronization is implemented in the WordRecord class to ensure that no 2 threads will access the variables at the same time. This is extremely necessary in this class since all the methods can be accessed by different threads simultaneously.

Code Safety

Thread safety is ensured by making each thread as independent as possible which minimizes the chances of multiple threads accessing the same data and thus ensuring there are no race conditions. It enforces single threading through modifying the shared resource in the critical section.

Liveness is guaranteed by the freedom from deadlock and starvation due to synchronization.

Deadlocking will not occur since the threading for the calculations is synchronized

Validation

Since word threads are generated from the same file, if the same word appears on the screen the score should only be updated with one instance of the word instead of every single

occurrence. This potential race condition was averted by updated the score through the text field action handler instead of the individual word threads.