**Class Description:**

There are 5 classes in my program with them being Score.class, WordDictionary.class, WordRecord.class, WordPanel.class and WordApp.class. They’re all stored in a package called skeletonCodeAssgnmt2;

**Score class**: The score class contains atomic class variables that record the number of words caught and missed. They’re used to store the score and the score, caught, and missed will get updated as the words are either reached the finish line or they’re correctly typed before they reach the finish line.

In this method, there are 7 methods, **4 get methods** for getting values of words caught, missed, total and score, we normally use this method to display the values in the WordApp. **2 increment** methods, one for increasing the number of misses, and the other one for increasing the number of words caught and update the score with the number of words caught. these methods are normally used when the words reached the finish line and needed to be updated before the next word appears. The last method is the **reset** method. We use this to reset every value in the score class, which is normally used when a game is completed. I have made no changes to this class.

**WordDictionary class:** This WordDictionary class is responsible for storing all the value some fruit names so that when the file cannot be found. The system can continue with WordApp without returning an error.

There are **two constructors** in this class, one with a string array as the parameter. This constructor will update the current dictionary with the input string array so that the program can work with what the text file provided. The other one with no parameter will allow the WordApp to work with the default dictionary. There is a **getNewWord** method that is synchronised. This method is used to add a new word that can be loaded onto the screen while running the app. The method is synchronised we want to avoid two different threads using the same shared memory. This will prevent that the app from possibly load multiple words onto the screen after a caught or missed. This ensures that only one word will be loaded onto the screen after each caught or miss. I have made no changes to this class.

**WordRecord class:** This is a class that stores all the data (such as falling speed, name, dropped, x coordinate and y coordinate) related to a single word in the dictionary. This class is mainly used to providing data necessary for the falling of words.

There are **3 constructors** in this class. The first constructor will receive no parameter, this is the constructor that sets all values to its default values, the default values are x and y to 0 and maxY to 300, dropped to false and generate a random falling speed that is independent of other texts. The second constructor will have one parameter, this constructor will call the first constructor which will make all values to their default values and change text to the text in the parameter. The third constructor will have a parameter with an X, maxY and text. Just like the second constructor, it is just that in this case, we change the value of x and maxY to the values given by the parameter.

All methods in this class are synchronised, this will prevent any 2 threads from performing any changes to the shared memory simultaneously.

There are a **setY, setX, setPos** and **setWord** method that changes the value of Y, X, the position of the text, text in the WordRecord class respectively. There is **getWord, getX, getY, getSpeed** methods that return the value of the text, X, Y, falling speed respectively. **resetPos** method is used to set the position of the word to its default position, which is y = 0. **resetWord** method is used either when the words are caught, missed or end of the game. This is for getting a new random words that will be displayed on the screen with random falling speed, dropped to false and y = 0. **matchWord** method is used to check whether the input word is the same as the text in **WordRecord** class. The Drop method is to increment y value with the value in the parameter. The value in the parameter represents the falling speed. The Dropped method would return a Boolean, true if the word reached the finish line and false if not.

**WordPanel class:** This class is responsible for the GUI component of the app. So, the colours of the frame, the font, font size, the falling of the words are all created in this class.

There is one **constructor** in this class that takes a string array and integer for maxY. Which sets all values related to the parameter to the value from the parameter. A void method called paintComponent, this method is for all the GUI setups, such as font, font size…

I filled in the **run** method, which is incredibly important for starting the threading and falling of words. First, I created a while loop and make a thread to sleep. Then create a for loop that runs length of the array number of times so I can change the falling speed for each word in the array. Repaint it so that we can see the movement of the words falling. If the word reaches the finish line, then I will reset the word, increment the missed words, and update the number of missed on the App. The run method will end when the number of words fallen is equal to the total words in the score class.

**WordApp class:** This is the main program that generates the app for the assignment. This class is mainly responsible for setting up the GUI, buttons, captions.

There are 4 buttons in this program:

* **Start**: This is used to start threading of the program, this is when we see the game starts and playable by the user
* **Pause**: This is used to pause the game by making done to true which will stop the threading until done is make into false.
* **End**: This will end the game. It will change the caption of caught, missed and score to 0. The falling words will be updated and moved to the top of the frame.
* **Quit**: This is used for quitting the game.

In the **setupGUI** method. This is where all the frame, captions, position of the words, buttons and panel are setup. The **getDictFromFile** will receive one parameter being the name of the file. This will read everything in the file and store them in an output array and later return the output array. The **main** method will read the input from args, using the input from args to update the WordDictionary and WordRecord. Then use setupGUI to generate the GUI for the user. The last few lines are for setting up the falling words for the GUI.

All these classes are used so that the app can run in accordance with the concurrency rules.

**Concurrency features:**

**Score class:**  There are 3 **Atomic Variables** here: missedWords, caughtWords, gameScore. We can use them just like the normal volatile variables, but unlike volatile methods, they can prevent deadlocks so it is obvious that atomic variables should be used in concurrency. Atomic variables will prevent any kind of race issues because it provides some operations that other threads cannot interfere.

**WordRecord class:** All the methods in this class are **synchronised**. This allows only one thread to use the method and edit the values of the class at any time. This is to prevent any data lost due to incorrect order of execution due to methods not being locked.

**WordPanel class**: There is a **volatile variable** called done. This variable should be a volatile variable because we are accessing it from the main memory. The reason we cannot make this an atomic variable is because this value must be updated in real-time. Therefore, any modification by any thread must be recorded in the main memory. Otherwise, there will be some runtime errors in the program.

**WordApp class:** There is a **volatile variable** called done just like the one in WordPanel. This variable should be a volatile variable because we are accessing it from the main memory. The reason we cannot make this an atomic variable is because this value must be updated in real-time. Therefore, any modification by any thread must be recorded in the main memory. Otherwise, there will be some runtime errors in the program.

**Ensuring codes:**

**Thread safety:** To prevent two or more threads from accessing the shared memory at once, there must be precautions to be done to prevent. This precaution must only allow one thread to access the shared memory at a time.

I used volatile variables, synchronised methods, and atomic variables to prevent such mistakes from happening. Volatile variables are used so that some values can access the main memory so that other threads can also use it. Synchronised methods are used to prevent different threads from accessing the same object at once, which will prevent data loss. Atomic variables are used to allow increments in the score to be thread-safe.

**Thread synchronisation:** Thread synchronisation is used to ensure that two or more threads do not execute some part of the program simultaneously to prevent data loss because of the wrong ordering of running the program. I used this in the score class because I do not want two or more threads to be accessing and changing the values in the score class. Without synchronisation, two threads can access the score class and one might change the value before another but the other one will change the value again without including the changes done by the first thread. This problem could change the recording of the score program which could lead to problems such as falling more than the expected number of words.

**Liveness:** This is referring to whether a concurrent program can start and finish in the expected time. The program is running as intended because of the right usage of the concurrent features of the program. The synchronisation, volatile and atomic features prevent any recursions, deadlocks in the program, therefore, prevents any delays that could happen to the program.

**No deadlock:** Deadlock is when two or more thread is holding onto their lock while waiting for the other thread to release their locks, but the other thread cannot release their locks because other threads are holding their lock which formed a cycle prevents any thread from releasing their locks. This is prevented in my program. However, deadlocks should never happen to a program with only one lock. This program only has one lock therefore no deadlock prevention measures are used in this program.

**System validity:**

To test whether this system is valid, the first and foremost method is the running of the program. Since the majority of parts of the program are given and I assume they’re correct. As I was using the app, the app showed no symptoms of wrong recording of caught, missed or scored. The start, pause, end, and quit buttons are all worked as intended. The speed of the words falling are different and all words will be counted as a miss as it reached the bottom of the app.

To do further testing, I tested the apps with multiple text files from small to large to check whether the codes work using either default or text files and there are no symptoms of the app not being responsive in either case.

The number of words on the screen and the total number of words that will be fallen are also tested with different values, from small as 2 or large as 50. There are no symptoms of the app not being responsive.

I also used different IDEs (JGRASP and Eclipse) and different OS’s (LINUX and Windows) to test this, and it worked for all IDEs and OS’s mentioned. Therefore, I assume that it should work for any OS and IDE’s.

Through all this testing, I will assume that this program will work in most cases. Bugs could still appear in occasional cases, but further tests are required to find such bugs.

**Model-View-Controller:**

The MVC is a computer architecture that separates an application into three main logical components: The model, view, and controllers. The view is the GUI that the program generates. The user can see the view which then the user will use the controllers, in this case, is the input from the user. The controller will be manipulated by the model which is comprised of classes. These classes will then update the view and the process repeats. This is the overview of the MVC for any program.

As from my program. The view will be the application, I can see the app and the words are currently not running. I can use the controls which is the keyboard and the mouse. I can click the start button and type on the keyboard. This is the use of controllers. The controller will manipulate the model which is my program in this case. So when the program receives the input from the controller, it will do the concurrent processing and then update the views which in this case is the application on my laptop screen. This process will continue until the user closes the application.

**Additional features:**

**Start Button:** I created this feature the user can increase the difficulty of the game by pressing the button multiple times. The falling speed will increase if the start button is press more than one time.

**Pause Button:** I created this feature where the user can pause their current game and can continue playing the game if he presses the start button.

**Info Button:** I created this feature where the user can press the button and a new frame will pop up. This new frame will contain a message that gives info to the user about how the game works.

**Preventing cheating:** To prevent cheating, every time the user press starts, the text box will automatically be cleared in case that the user tries to pause, type the words, press start and enter without typing the words while playing.