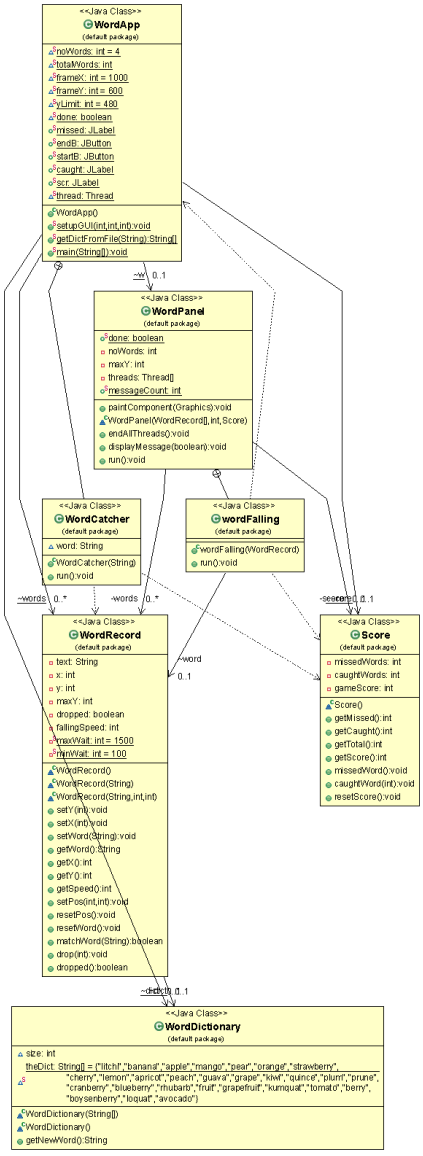
Assignment 4 Report
Shai Aarons (ARNSHA011)

1. Class Structure and Descriptions

In order for the typing game to run as a fully concurrent program, the class structure needs to be developed in such a way that will allow for the process of multiple threads performing different operations, while working together. In the following class structure, we have multiple different operations, enclosed in their own threads, performing vastly different operations. Whilst discussing using locks within the java program, we use the words synchronized and locking interchangeably. Each class, as well as their functionality, will be discussed below:



1.1. WordDictionary Class

This class merely functions as essentially a dictionary of all the words that will be displayed and fall on the screen. Here an array stores all the words that will be displayed. In the case that the user does not input a text file name, there is a default set of words. The class has two instance variables: A dictionary array that stores a set of words (in the form of a string) as well as an integer value representing the size of the array. The class has a parameterized constructor that receives a set of words and stores them in the dictionary. The class also has an important method called getNewWord() that returns a random word within the dictionary to be displayed on the screen. This uses the Math.Random function, which provides sufficient enough randomization of the words. The getNewWord() method has the synchronized keyword attached to it. This will allow for sufficient concurrency in terms of multiple threads accessing the base of words at the same time, without causing any bad interleaving.

1.2. Score Class

This class represents the score of the user whilst playing the game. This class stores an integer called missedWords, which represents the amount of words that the user did not type in quick enough time before the word reached the bottom of the screen. The class also stores an integer value called caughtWords, which represents the amount of words that the user did indeed type in time before reaching the bottom of the screen. The final instance variable that the class holds is an integer called gameScore, which represents the score of the user - this is incremented by the length of any word caught in time by the user. Within the class there are accessor and incremental methods for each variable - these are all synchronized methods. The reason for this is so that the blocks of code within the methods happen as an atomic unit in order for efficient concurrency to occur and so that bad interleaving can never occur when multiple threads are trying to change the same shared variables at the same time. There is also a

synchronized Method to increment the missed word counter by one when the user does not type a word in time or alternatively a synchronized Method to increment the caught word counter by one when the user types a word in time (as well as a score by the length of inputted text). There is also a synchronized method to set all the instance variables to zero when the game is over or restarted

1.3. WordRecord Class

This is the class that represents a falling word, as well as its associated animations, within the panel of the program. This stores a string value that represents the actual word that is falling. It also stores both the x and the y positions of the word relative to where it is displayed on the screen. There is also an instance variable integer that represents the furthest point the word can appear within the screen, any point beyond there the word can be considered as missed. There is also a Boolean variable called dropped which holds the value 'true' if the word has begun to fall on the screen. There is also an integer called fallingSpeed, which holds the speed at which a word is falling. The final variables that the class stores are the variables that store the minimum and maximum amount of time before a word begins to drop from the top of the screen. Within the class there are accessor and mutator methods for each variable - these are all synchronized methods. The reason for this is so that the blocks of code within the methods happen as an atomic unit in order for efficient concurrency to occur and so that bad interleaving can never occur when multiple threads are trying to change the same shared variables at the same time. There is a method called resetWord that is a synchronized and locked method that occurs when a word has been either caught by a user or missed by the user. The position of the word is set back to the top of the screen. The word is then randomly reselected from the WordDictionary class. The method also sets the falling speed to a random value. Within this method there is effective concurrency measures as it is critical that this block of code happens as a singular unit, without other outside sources changing the data at the same time. This is also a synchronized/locked method called matchWord that checks if the word the user has typed is the same as the word that is stored in this class. There are also further measured that we implemented to ensure that this class allows for the animation to occur effectively without any potential issues of deadlock

1.4. WordPanel Class

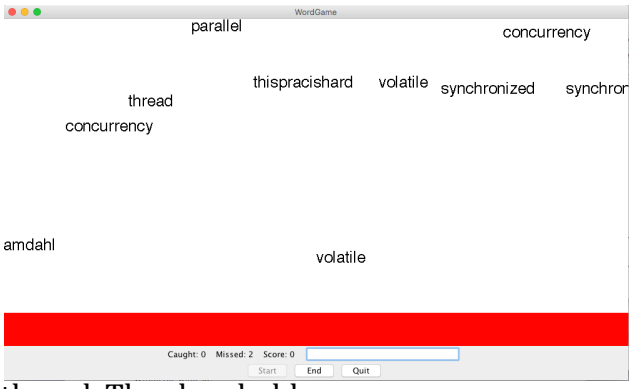
This is the class that represents the panel whereby the words will be falling. This is essentially a placeholder for all the WordRecord instances that will be falling. This class will be used by the WordApp class to essentially represent the graphical user interface that occupies the majority of the screen. The class extends and inherits from the JPanel class and implements the runnable interface so that it can be used an independent thread.The class holds the following important variables:

- The volatile Boolean method done that stores the value if the game has been completed or not and the volatile keyword is used as an indicator to the Java compiler and Thread to not cache value of this variable and always read it from main memory.

- An array of WordRecord instances that are the array of words (and their associated animations) that fall from the screen
- An instance of the score class to hold the users score whilst playing the game
- An array of the Thread class that hold the individual words falling as independent threads
- An instance variable called message count that stores the amount of times a success or fail message has been displayed to the user

This class has an inner class called WordFalling that serves as individual threads of singular words dropping from the top of the screen. More on this can be found below. There is a method called endAllThreads() that is the method that will end all of the threads when a user wishes to end the game. This iterates through all the individual threads and uses the join() functionality to ensure all the threads have run enough computation in order to end the game. There is also a method called displayMessage() which is a synchronized method that will display a message to the user that shows if the user has won or lost the game as well as the users score for that iteration of the game. This message will only be displayed if the messageCount instance variable is equal to zero. The messageCount instance variable will be incremented every time the method is called, hence why the method is synchronized due to multiple threads potentially accessing this method at the same time. There is also the run method for the panel of the GUI. This method creates and starts all the threads of wordFalling threads using the threads array and the start() method.

<u>1.5. wordFalling inner class</u>



This is the inner class that represents a singular word falling within the panel on the graphical user interface. This implements the runnable interface so that it can be implemented as an individual thread in this multithreaded game.
Many instances of this class will be created by the WordPanel class so that there can be multiple independent words falling from the top of the screen, all enclosed within their own thread. The class holds a WordRecord instance so that it can manage the word falling from the screen. There is also the run() method that will be activated once the panel starts this as an individual thread.
This thread has a while loop that will essentially run until the game is over. Within the while loop, the following process occurs:

1. If the word has dropped the missed word counter is incremented and the associated messages are displayed on the WordApp. The word is then reset using the .resetWord() method
2. The word will temporarily sleep using the .sleep() method
3. The word will then drop by a certain amount - using the wordRecord.drop() functionality
4. If the game is over a message will displayed

1.6. WordApp class

This class is the class that essentially functions as the main class or driver of the program. This class sets up the graphical user interface and implements the associated classes and the WordPanel class. This class will allow the user to start the game by clicking a button that will start the WordPanel thread, as well as its individual wordFalling threads. The user can end the game by clicking a button that will end all the panel thread as well as all the individual threads of words falling. Every time the user types a word in a text box and presses the enter key, the program will spawn a new WordCatcher thread that will evaluate if the user successfully typed a word or not. The user can also exit the program by pressing the 'Quit' Button. This class has an inner class called WordCatcher that serves as individual threads that will evaluate if the user typed a word correctly or not. More on this can be found below. There is a method called getDictDictFromFile that opens the file and reads in the amount of words to be used by the game. The words are then stored into the WordDictionary class. There is also the main method of the program that will initialize the process of the game. It reads in the words from the text file and then sets up the graphical user interface.

1.7. WordCatcher Inner class

This is the inner class that evaluates if a string that the user has typed in matches to a word that is falling on the screen. This class implements the Runnable interface so that it can be used as a thread to perform its own operations separately from the rest of the program. An instance of this class will be created every time a user types the enter button in the main GUI. The Run method that will iterate through the array of WordRecord instances use the WordRecord matchWord() method to evaluate if the user has correctly typed in a word. If the user has correctly typed the word - the score will be incremented and the word will be reset using the resetWord() method.

2. Java Concurrency Features

With this game being inherently multithreaded in nature, many steps were taken in order to work as a combined concurrent system. With concurrency, this program needs to include mutual exclusion of certain blocks of code – this is done via locks. Locks ensure correct behavior of accessing and changing shared resources by multiple threads. For example: if two threads wish to acquire the shared data at the same time – one will ‘win’ and one will block. In java this is done by using a mechanism called a ‘Synchronized block’ which is a system for enforcing atomicity of code via reentrant locks. The synchronized blocks of code are incredibly useful as they are not released until the code within the block has finished its execution – thus it is impossible to not release the lock. In this project the main concurrent feature that was used were these synchronized blocks of code at the method level.

All the accessor, mutator and incremental methods within the score class were made to be synchronized (and hence locked). The reason for this is that the score will be updated and accessed by multiple different wordFalling threads at the same time – thus it is essential that locks are used in order to prevent a bad interleaving scenario whereby the score is not correctly updated or the wrong score amount is accessed.

Within the WordRecord class majority of the methods regarding positioning of words, as well as animations are made to be in a lock structure. This is in order for the words that are falling to actively avoid succumbing to bad interleavings which would result in faulty execution of the game.

In the WordPanel class there is a method called displayMessage. This was encapsulated in synchronized block as multiple threads will be accessing the counter of messages displayed at the same time. Thus it is essential to ensure that this counter can only be incremented by one thread at a time.

The java volatile keyword ensures that variables with said keyword are not cached and a read function always returns the most recent write by any thread. This keyword was used by the 'done' Boolean variable across multiple classes. The reason for this is so that the program can immediately determine when the game is over without certain threads 'believing' that the game has not yet ended.

No atomic variables or barriers were used within the game as the synchronized block supplied sufficient concurrency, with no chance of bad interleaving ever occurring. There was also no need for synchronized collections as the use cases for this project did not require the use of complex data types.

3. Concurrent Processes Within the Game

There are many subsections within the realm of concurrent programming that need to be considered whilst programming a game of this nature. The features processes implemented to ensure properly working concurrency will be discussed below:

3.1. Thread Safety

Whilst writing a concurrent program, it is essential to ensure that we are monitoring and managing an objects state – we need to protect shared variables from concurrent access by multiple threads. A class can be considered thread-safe when it performs its operations correctly when its variables are being used and accessed by multiple threads. Due to the 'synchronized' keyword used on majority of the methods regarding the shared variables, the classes within the program can be considered as thread-safe. There is never a need to add additional synchronization on the part of the calling code within the program. For each location where synchronization was used, there is a lock that is always guarding the location.

Furthermore, the Java Swing toolkit, which was used for the graphical user interface of the program, uses thread confinement extensively. The visual components and data model objects are not themselves thread safe, however safety is achieved by confining them to the Swing Event Dispatch Thread.

3.2. Thread Synchronization

Refer to section 2 for a detailed probe into how thread synchronization has been implemented in the program

3.3. Liveness

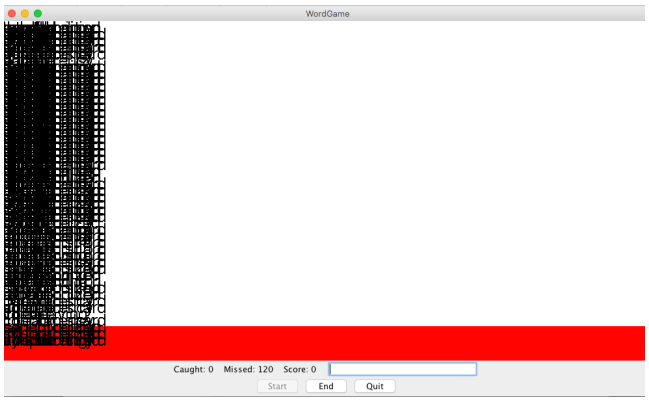
Liveness refers to the property within concurrent computing that require a system to make progress despite the fact that the execution of critical sections may have to alternate between multiple threads. The synchronized parts of the program have no serial counterpart. The program will continue in its execution due to the threads acting in parallel

3.4. Deadlock

Due to the way that this project is structured, there is no major possibility of deadlock to occur. However, despite this, the critical sections within the program were made to be as small as possible in order to perform multiple operations that may cause a 'deadly embrace.' More features, however, could have been implemented to further protect from an absolute worst-case deadlock scenario such as coarsening lock granularity – however due to time constraints this was unable to be implemented.

4. Validation of the System

The system was validated through a set of tests in order to test its performance in executing the required tasks. The system was evaluated by how many threads the program can handle at once – i.e. the amount of words that can be falling at a single time. The following observations were when performing these tests:



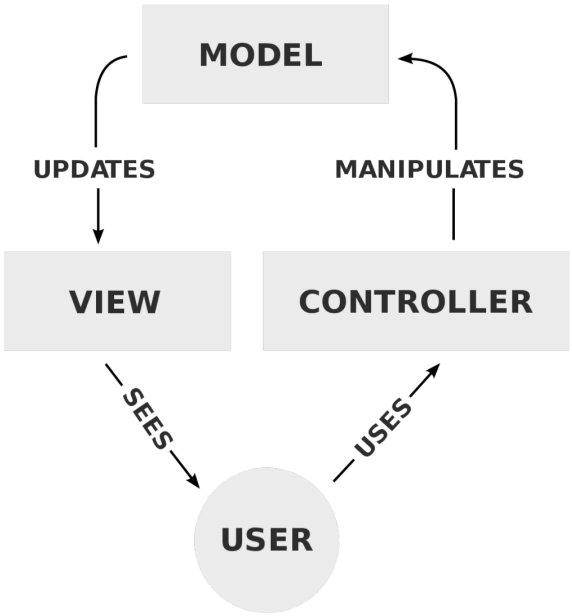
- For anywhere under 1000 words/threads the program behaves well and is a fully functioning concurrent computation with all functionality performing up to standard
- When the amount of words/threads are above 1500 the program still works, however it begins to behave erratically – with only one column of words being displayed at a time. This is most likely due to the overheads of managing a vast amount of threads, accessing the same shared resources and variables.
- When the amount of words/threads are over 2200, there is an outOfMemory error and the program does not work. This indicates that on a 4 core CPU with 8GB of DDR3 Memory, there is not sufficient storage space to allow for this massive amount of threads working together concurrently

The program was also tested for any potential race conditions. In this program there is very little to zero race conditions that have been displayed through extensive testing via actually playing and observing the games performance and operations. Through observations, the game performs well and there is no clear sign of any bad interleavings. This can also be due to the locks and reentrant locks that have been put in place to mitigate these exact errors. The java

synchronization features are incredibly useful for the avoidance of typical errors presented by concurrent computing

5. The Model-View-Controller Pattern

The MVC software design architecture/pattern has been implemented here in this project due to its graphical user interface. This is the software pattern commonly followed whilst programming for a project with a user interface. In this project this was performed through:



Model - The model here comprises of the classes WordDictionary, the array of WordRecords and the score. The text file holding the words can also be viewed as what makes up the model. The model is the part of the program that is involved with storing the data and providing access to it.

View - The view here is the actual graphical user interface itself. This is the GUI set up by the WordApp, WordPanel and WordRecord classes. The view includes all animations associated with the words that are falling.

Controller - The controller is the threads that alter, control and monitor the model and the view. These are the wordFalling and WordCatcher Inner classes that create and end the threads of words falling. These processes move the word positions, control the animations in the view, monitor the process of adding and removing (catching) the words as well as updating the score and various counters (missed and caught counters) within the program

6. Additional features/Extensions

One additional feature that was added was the functionality that allows for new threads to be created every time the user presses the 'enter' key. This new thread manages the process of determining if the word entered matches any of the words displayed on the screen. The reason this functionality was added was to demonstrate concurrency working at its full potential – we have certain threads which are in charge of executing the process of each word falling on the screen, an overall thread that deals with the display of the panel as well as these final additional threads which manages the process of 'catching' the words. Another reason this functionality was added is because the user/bot could potentially enter words at a rapid pace, hence needing multiple threads to handle previous instances of entered words, whilst spawning new threads of execution every time a word is input.

7. Appendix: Git Log

```
commit 5516e04fa6ac8c46d2fb96ea7143f3128f8a51d0 (HEAD -> master)
Author: Shai Aarons <shai.aarons21@gmail.com>
Date:   Sun Sep 29 16:27:16 2019 +0200

    Everything working.
    Assignment Finished.

commit 3e7c93036b9b7066954e4709a061739594eaa557
Author: Shai Aarons <shai.aarons21@gmail.com>
Date:   Sat Sep 28 11:46:41 2019 +0200

    Everything working. The word catching process is also now concurrent and happens on seperate threads.
    The next step is to fix the end button and start button
    Further optomizations will be looked into

commit 99be0475e15914835966e1c28abf4991f33e3253
Author: Shai Aarons <shai.aarons21@gmail.com>
Date:   Fri Sep 27 15:53:09 2019 +0200

    The game fully works and is functional. I am still confused about when the game should end however.
    The next step is to potentially have to words being caught in parallel if that can actually happen

commit 411405b189a3f5092533971b311486a519520e09
Author: Shai Aarons <shai.aarons21@gmail.com>
Date:   Fri Sep 27 13:45:39 2019 +0200

    Words are dropping. The start and stop buttons are working mostly. The quit button works.
    Next is creating a thread to manaage when the user types in a word and work with the other threads.

commit 7bc6140f74b2222854dca9e252d4086d1e233dd7
Author: Shai Aarons <shai.aarons21@gmail.com>
Date:   Fri Sep 27 10:11:12 2019 +0200

    First Commit
~
```