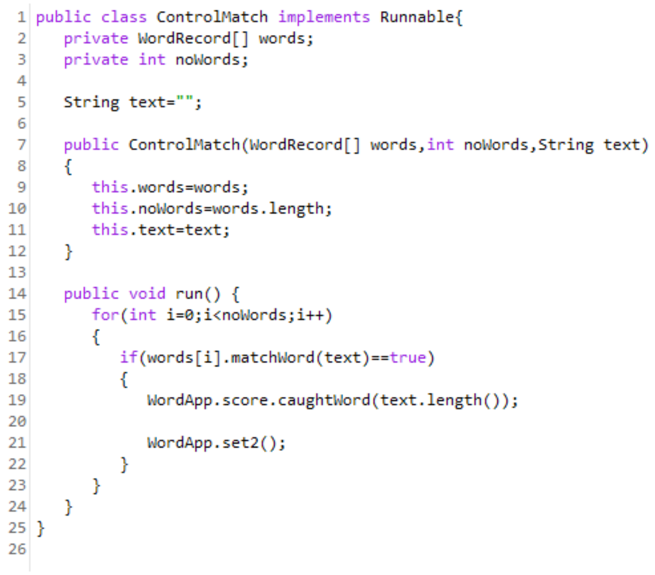
MPNSIZ005 **Assignment Two** **CSC2002S**

1. Added Classes

Why Thread Class Is Implementing Runnable?

This answer applies to all the created classes. In Java we have two options when we want to execute a piece of code on another thread. We can extend Thread class or implement Runnable interface and pass our runnable object to some other thread. Runnable is an interface which represents a task that could be executed by either a Thread or Executor or some similar means. On the other hand, Thread is a class which creates a new thread. The start() method creates a thread of execution which is attached to an instance of Thread class. Implementing the Runnable interface doesn't create a new thread. The run() method of Runnable is called making it execute the task on the thread of execution, and the start() method run quickly. You cannot run Runnable without the Thread class

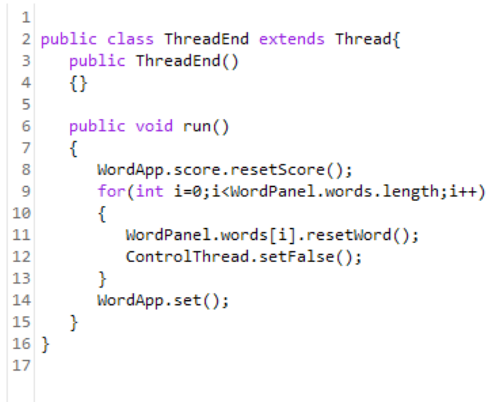
i) ControlMatch



```java
public class ControlMatch implements Runnable{
   private WordRecord[] words;
   private int noWords;

   String text="";

   public ControlMatch(WordRecord[] words,int noWords,String text)
   {
      this.words=words;
      this.noWords=words.length;
      this.text=text;
   }

   public void run() {
      for(int i=0;i<noWords;i++)
      {
         if(words[i].matchWord(text)==true)
         {
            WordApp.score.caughtWord(text.length());

            WordApp.set2();
         }
      }
   }
}

```

words is an array of all words; text are the words received through the text filed of the GUI as they are typed. This class matches the typed word to the ones on the screen. If they are the same it removes it from the screen and increase score by the length of the word. run() is called by a thread in wordPanel.java.

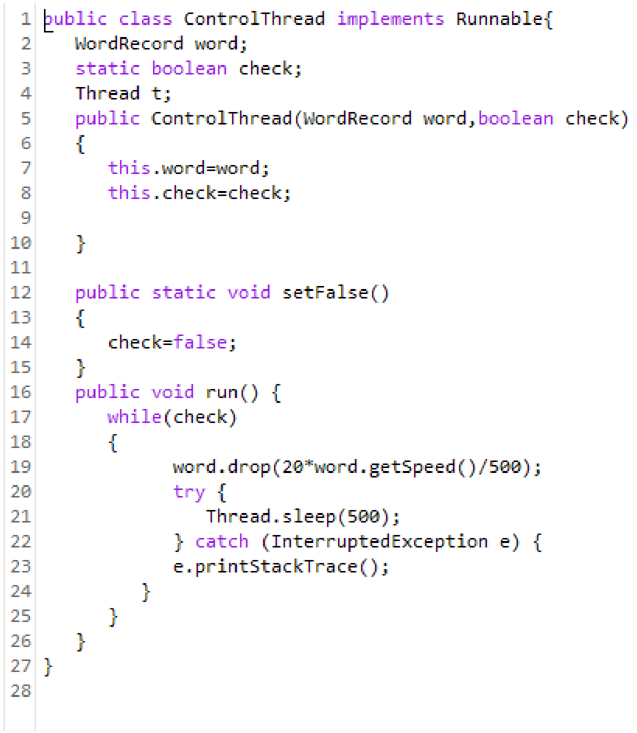
ii) ThreadEnd



```java

public class ThreadEnd extends Thread{
   public ThreadEnd()
   {}

   public void run()
   {
      WordApp.score.resetScore();
      for(int i=0;i<WordPanel.words.length;i++)
      {
         WordPanel.words[i].resetWord();
         ControlThread.setFalse();
      }
      WordApp.set();
   }
}

```

This class is responsible for terminating the falling threads (words falling) safely on the click of the end button. run() is invoked the moment end button is clicked, it then resets the scores back to zero using resetScore() method in the Score class. The for-loop is for stopping each word falling and is safely closing the thread of setting the 'done' variable to true.

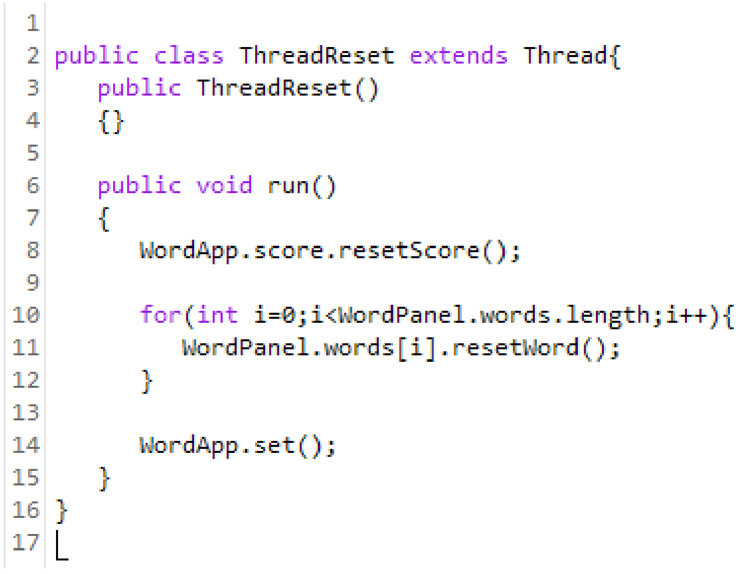
iii) ControlThread



```java
public class ControlThread implements Runnable{
   WordRecord word;
   static boolean check;
   Thread t;
   public ControlThread(WordRecord word,boolean check)
   {
      this.word=word;
      this.check=check;

   }

   public static void setFalse()
   {
      check=false;
   }
   public void run() {
      while(check)
      {
            word.drop(20*word.getSpeed()/500);
            try {
               Thread.sleep(500);
            } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }
}

```

This class controls the dropping speed of the words and the sleeping time of those words on the screen. run() will be called whenever the thread gets started, it has the implementation of what the created thread does. setFalse() stops the dropping thread safely because java thread.stop() does not terminate the thread safely, it might result in a deadlock. The speed is the word that is falling multiplied by 20 and all divided by 500 to prevent illusion. The thread only sleeps for 5 milliseconds.

iv) ThreadReset



```java

public class ThreadReset extends Thread{
   public ThreadReset()
   {}

   public void run()
   {
      WordApp.score.resetScore();

      for(int i=0;i<WordPanel.words.length;i++){
         WordPanel.words[i].resetWord();
      }

      WordApp.set();
   }
}

```

This class starts the game from scratch. run() is invoked when the restart button is pressed, it then restarts the scores and the caught word to zero, it is also responsible for clearing the words falling and then start them at the top of the screen.

2. Updated Classes and the Explanation of the Modifications Made

(1) wordApp.java

First the variables of the GUI were changed to static so as to make them to be class variables over method variables so they can be refenced across the whole wordApp class. A thread was created in the in this class after the click of the start button and it will start the run() method of WordPanel.java. This does not violate the MVC properties because the thread creation is happening outside the class. The threads are created after each button is clicked (start, restart, end and quit), this is useful for starting a thread and make it alive.

(2) Score.java class

This class did not require much changes so the only modification is the synchronization of the getters and setters. This is to prevent bad interleaving especially data race, by doing this the correctness of the game is increased.

(3) WordPanel.java class

The changes were only made to the run() method. The implementation of this method is to control the animation of the words at the screen. This method is also responsible for creating the thread of the words and putting them in alive state, this is useful because it is the basis of our game.

3. Description of The Concurrency Features and Their Importance

In this assignment only, the synchronized keyword was used. When more than one threads are ran, we may encounter a case where a number of threads try to access the same resource and that can result in some concurrency issues. Like if multiple threads try to write at the same memory location, one thread may damage the file because one of the threads overwrite, or while one thread is opening the file another thread may be simultaneously trying to close it.
So that is why we synchronize the action of multiple threads and make certain that only a single thread can access the resource at a given time. This implementation process is called **monitors**. Every java object is associated with a monitor, which a thread can lock and/or unlock. One thread holds a lock on a monitor at a time.
Java comes with a way of creating threads and synchronizing their tasks by using synchronized blocks. Shared resources are kept within this block. We can also have multiple threads try to update the score simultaneously in the score class, this can give out free scores incorrectly. Also, with the misses, if more than one thread attempts to write on the variable, misses, it will result in this variable being updated twice and an incorrect reflection on the missed words.

4.

4.1. Ensuring Thread Safety

Thread safety – a class is considered thread safe if it behaves correctly when accessed from multiple threads, regardless of scheduling and interleaving by the runtime environment and with no additional synchronization on the part of the calling code, i.e., a thread safe class cannot be placed in an invalid state. In our code this was prevented by using static methods in the EventQueue class – invokeLater() and invokeAndWait(). These methods take a thread as a parameter and are responsible for executing the thread that is passed in synchronization with the Swing main thread. They determine how the update should occur, the invokeLater() method returns immediately, placing the update code in the regular event of Swing. Sometimes we may want to wait until an update has occurred, in that case we use the invokeAndWait which does not return until the update is complete. The two methods handle the threaded Swing code.

4.2. Ensuring Thread Synchronization

Threads occupy the same memory location meaning they share resources. Sometimes we want a single thread to access a shared resource at a time. Java has synchronization for this type of situation to control the access. In the absence of this, we may have one thread attempt to modify a shared object's value which leads to an error. For example, Score could be updated by multiple threads leading to unexpected and unwanted results, same for the misses variable. Thanks to java synchronization by including the synchronized keyword in our methods and variables this is avoided.

4.3. Liveness

Liveness refers to a set of properties of concurrent programming, that require a system to make progress despite the fact that its concurrently executing processes may have to take turns in critical sections (access of shared resource). The isAlive() function is used to check to check if a thread is alive or not. Alive refers to a thread that has begun but not been terminated yet. When the run method is called, the thread operates for a specific period of time after which it stops executing. It returns true or false which our code uses to keep track with the liveness of each thread.

4.4. Avoiding Deadlock

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. For example, process 1 (or thread 1) is holding resource 1 and waiting for resource 2 which is acquired by process 2 (or thread 2), and process 2 is waiting for resource 1. In this project we tried to avoid deadlock by using the following ways:-
-Avoiding unnecessary locks: locks were used only for those members on which it is necessary
-Avoiding nested locks: we tried as best to avoid giving a lock to multiple threads if a lock is already provided to one thread.
-Using lock ordering: always assign a numeric value to each lock. Before acquiring the lock with higher numeric value, acquire the locks with a lower numeric value.
-Lock time out: we can also specify the time for a thread to acquire a lock. If a thread does not acquire a lock, the thread must wait for a specific time before retrying to acquire a lock.

5. Validating System and Checking for Errors

First, we look and understand the code if we found nested synchronized block or trying to get a lock on a different object or calling a synchronized method from other synchronized method, these reasons lead to a deadlock situation. Another way to detect is to use the **io** portal. It allows us to upload a thread dump and analyse it. We also used the **jConsole** or **VisualVM** to detect multithreading errors. It shows us which threads are getting locked and on which object. To try and prevent race conditions we can enforce single threading through methods that modify a single shared resource. For validation we used the synchronized keyword where multiple threads write at the same time. We also used atomic variables to make sure that if a variable gets updated it should be visible to all running threads.

6. Model-View-Controller Pattern of Our Code

This pattern is used to separate application’s concerns.
-Model: model represents an object or java POJO (Plain Old Java Object) carrying data. It can also have logic to update controller if its data changes.
-View: view represents the visualization of the data that model contains.
-Controller: controller acts on both model and view. It controls the data flow into the model object and updates the view whenever data changes. It keeps view and model separate.
In our project we have WordApp object acting as the view because it has the GUI implementation and is responsible for all visualization of our game. WordPanel is our controller, it accepts input from user and other classes and converts it to commands for the model and view. Our model is the class which manages the data (words) which is the WordRecord class. It directly manages the data, logic and rules of our application. It receives user input (<total words> <words on screen> <dictionary file>) from the controller. Then the controller (WordPanel.java) responds to the user input and performs interactions on the data model objects, then passes the input to the model.