Computer Science

CSC2002S

Assignment 2

Java Threads Module

Concurrency

Josh di Bona (DBNJOS001)

31/08/14

Concurrent programming is to correctly and efficiently manage access to shared resources as opposed to parallel programming extra computational resources are used to solve a problem faster. The focus is on task decomposition and not data decomposition. Threads are assigned different tasks and access the same resources. The fact that the threads now access the same resources instead of having their own means that we have to protect our data from events such as deadlock.

In the following report ill describe and explain the coding I have done to create the game and the steps I took to protect the data.

There were no extra classes added to the provided skeleton code, but modifications were made. The only class that remained unchanged was WordDictionary.java. In WordRecord.java I made the class runnable therefore also gave it a run() method. The run method moves the word 20 pixels down the screen at a randomly generated speed. A shared Boolean flag was added to state whether the word was currently visible in the screen or not and methods to go along with it, namely checkVisible(), makeVisible(), makeInvisible().

In WordPanel.java I started with importing the necessary packages needed to deal with images in order to create the background image in the game. I added a few more variables almost all of which are shared with WordApp.java. A count for the number of words that have fallen so far, score object, 4 JLabels, caught, missed, message and scr. Max for the max number of words which are allowed to fall during the game. In order to put an image in the background of the game, I stopped the clearRect(),setColor() and fillRect() methods and added a drawImage() method call in their place. All which is under paintComponent. I added code here to check for any words that have reached the bottom of the screen and need to be replaced, I update all my JLabels and draw all the words in the words[] array. I initiated some more variables in the constructor and because the class implements runnable, it has a run() method. In here I added a while loop to constantly call repaint() then sleep a certain amount of time in order to animate the movement of the words.

In WordApp.java I made a new JLabel to display a message at the end of a game, as well as a new JButton to handle the quitting of the game. I then modified all the actionPerformed events for the start and end buttons, having the start button start the game (by starting the necessary threads), as well as start a new game when clicked again (after either pressing start or end). The end button will have the threads stop moving the words and create new words, and not do anything until the user presses start again. Which will reset all the previous game scores and information.

In order to make the code concurrent and to protect the shared data, I used a few concurrency features. The first feature being volatile variables, I used then for shared variables such as the Boolean done and the integer count in WordApp and WordPanel. I used volatile in this case as it can be used in very simple cases such as flags in order to only allow 1 thread to change it at a time and helps to prevent possible deadlock and even a race condition.

The second concurrency feature I used was synchronization, I synchronized all the getters and setters in the score class as well as the WordRecord class to make them both a thread-safe class. This was necessary because WordRecord is the model in the model view controller and is accessed by many threads. No two threads can access the same shared data at the same time, and with this type of game, that is very easy to happen and you could end up with race conditions and stale data. For example if you type in a word while another word hits the bottom, or two words hit the bottom at the same time and now two threads are both trying to implement the missed word count.

I ensured thread safety for both shared variables and the swing library by synchronizing the access to the shared data. You need to protect data when there is more than 1 thing that can access it at any given time, whether it's for reading or writing. This is because of stale data and race conditions. A race condition occurs when the output/result of a process is unexpectedly and critically dependent on the relative sequence or timing of other events. The memory accessed by threads is non-deterministic and they race each other to influence the output first. The class (WordPanel) which handled all the drawing and changes to the GUI was a thread on its own. Any code that accessed the GUI ran in the event thread.

I ensured thread synchronization (mutual exclusion – event A and B must not happen at the same time) by creating a thread for every word on the screen, so if 3 was given as the command line parameter for the max number of words to be on screen at any given time then 3 threads were created to handle their movement. They all access the same shared data with synchronized getters and setters.

In order to ensure liveness (a concurrent application's ability to execute in a timely manner) any code that accessed the GUI had to be done through the event thread. Each word had its own thread moving it in so that you don't have to wait for 1 thread to execute all the movements before the program can do anything else.

In order to prevent deadlock thread synchronization was used for all the getters and setters in Score and WordRecord, which is where many threads might try and access at once at any given time. Synchronized methods have locks and release them by themselves to help prevent deadlock.

In order to check for any possible race conditions and to validate my code, I removed the randomly generated speeds and replaced it with a set speed in order to check that when more than 1 word hit the bottom at exactly the same time, the missed word count would count all 3 and not just 1. I changed the fallingSpeed in WordRecord to 250 and waited for all 3 words to hit the bottom at the same time, and made sure that the missed count went up by 3.

I also checked that everything happens correctly if you enter a word while at the same time a word hits the bottom and the score, missed word, and caught count must all be implemented, and it does.

My design conforms to the Model-View-Controller pattern with the model, which accepts a command to change state (increment a counter) and emit notices of change of state and the new state as WordRecord which holds all the shared variables and states of the words as well as methods to get and set variables (or the states).

The view is the UI(or the GUI) which shows the model's state and allows the user to enter commands which will change the model's state. In my case the view is WordPanel, which has buttons and text fields which allow the user to communicate with the GUI and show its state (the words position on the screen, the score and other variables in the text fields).

The Controller is a listener, it has the actionPerformed() methods and increments the values in the model. It sends a notice to the console, initialises the model and has methods to get references to the model and the view. In my case this is WordApp, which has all the actions for the buttons and bridges communication between the model and the view.

There are many improvements that could be made on the game even with it being an educational typing game…but unfortunately all I had time for was to make it look a little better with a nice background.