**CSC2002S ASSIGNMENT 2**

**CONCURRENCY**

**MFNLIN003**

**Introduction**

The aim of this assignment is to design a multithread java program that ensures thread safety and concurrency for the application to function well. To archive thread safety synchronization was used which is capable to control the access of multiple threads to any shared resource. In the case of the assignment this was implemented to enable a transfer of water and pausing of the process to happen separately.

Concurrency

 Is the ability to run several programs or several parts of a program in parallel. If a time-consuming task can be performed asynchronously or in parallel, this improves the throughput and the interactivity of the program

Multithreading

 Is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process. Threads are lightweight sub-processes; they share the common memory space. In Multithreaded environment, programs that are benefited from multithreading, utilize the maximum CPU time so that the idle time can be kept to minimum. Threads can be created by using two mechanisms

1. Extending the Thread class
2. Implementing the Runnable Interface (this is the way implemented in the assignment)

**Expected Application Behavior**

To archive the assignment objective further develop the parallel programming application from assignment 1 to contract a grid terrain using the text files containing details of the terrain. The application is supposed to make using of the Swing class to contact a Graphical User Interface (GUIs). The GUI is supposed to display the grid terrain landscape image as a greyscale image. By clicking on the window water should be added to the terrain as an overlay of the terrain image. By clicking the play button, the simulation of water is supposed to be evoked. Water is transferred to the lowest point among the surrounding points. When the pause button is pressed the simulation is supposed to pause. When the rest button is pressed all water is to be cleared from terrain and the end button disposes the window.

**Classes modifications and Methods**

Several modifications were made to the *FlowSketon* package classes in order to archive the objective of the assignment. A new class Water was created. Below are the detailed explanations of the modification.

1. Terrain Class:

The only modification added to the Terrain class is the *getHeight()* accessor method. The method is used to retrieve the terrain elevation at a given point based on the coordinates passed as parameters.

1. Water Class:

The Water class is used to add water to the grid terrain by creating a buffered image with blue colored pixels and overlaying it on the greyscaled terrain image. It is also used to transfer water to the lowest surrounding point.

The class has instance variables *t* (Terrain object), depth (depth of water added to each point) and *water* (Buffered image of water to be added to grid terrain). The Water class constructor has the same name as the class name. Its purpose is to initialize the object of the Water class by executing the java code and the buffered image by setting it to the same size the terrain image. The permute array list from the terrain class is populated with linear index positions to allow random traversal over terrain during water transfer.

*addWater()* method it is a setter method since it does not have a return type. It takes the coordinates of point as parameters. First check if water has not yet been added to that point already by making use of the *BufferImage.getRGB(x,y)* method that returns the integer value of the color at that pixel point. If the value does not meet the blue color int value, then water must be added in the 3 by 3 area of the point by coloring every pixel on water image blue using a loop.

*convertWater()* method it is setter method since it does not have a return type. It is used to convert the unit of water to its double value as one unit represents 0.01m of water.

*getWaterSurface()* method is accessor method since it returns a float value. The method is used to return the water surface which is the grid elevation plus the water depth at that point.

*transferWater()* method is a setter method since it does not have a return type. The method is used to iterate through the Permute array list and use the *getPerumate()* to get 2D coordinates of a point. The color pixel of the point is checked if it is blue which means there is water on that point. If so, water is supposed to transferred to the lowest point surrounding the current point by coloring the current point transparent, removing one unit of water from it, coloring the lowest point to blue and coloring the lowest point pixel blue. This is archived using of the *BufferImage.setRGB()* method.

*getImg()* method is an accessor method since it returns the water image.

*removeWater()* is a setter method since it does not have a return type. The method is supposed to remove all water from terrain by coloring all pixels in the water image to translucent. This is achieved using *Color.TRANSLUCENT* and *BufferImage.setRGB()* methods.

1. FlowPanel Class:

A couple number of modifications were made to this method below is a detailed description to the modifications.

The instance variable in the class is now the Water object (*water*) instead of the Terrain object. This change was made to ensure that the water methods like *transferWater()* and the getImg() method in order to transfer water and overlay the water image over the terrain image. Two AtomicBoolean variables are created named *running* *and paused.* The variables are used to control the water transfer simulation. A private final Object is created to ensure thread safety by the locking and unlocking of threads when pause and play button are pressed. An AtomicInteger is declared l*ock*  that is going to count the number of timesteps it takes the flow threads to complete the water transfer simulation. The AtomicBoolean variable lock is used as a flag to indicate that the lock (*pauseLock*) has been locked or not.

Flow panel constructor has the same name as the class used to initialize the Flow Panel object. By intialising the Water object, the Object *pauseLock*, atomicInteger by setting is to zero, the AtomicBoolean variable *lock* by setting it to false, the AtomicBoolean variable *running* by setting it to false, the AtomicBoolean variable *paused* by setting it to false

*paintComponent()* method the only modification in this method is by calling the *Water.getImg()* method that returns a bufferedImage of the water image and drawing the image to the graphics component by overlaying it on top of the terrain image passed in the water class .

end() method is a setter method that sets the AtomicBoolean running variable to false (*running.set(false*)) .This method is used to terminate the water transfer simulation and unlock the synchronized lock by calling the *unpause()* method.

*pause()* method is a setter method that sets the AtomicBoolean pause variable to true (*paused.set(true))* .This method is used to pause the water transfer simulation.

*unpause()* method is a setter method that sets the AtomicBoolean running variable to true (*running.set(true))* and setting the AtomicBoolean paused to false (*pause.set(false*). This method is used to resume the water transfer simulation, by unblocking the thread and unlocking the lock by calling the *unlock()* method.

*step()* method is a setter method it checks if the AtomicBoolean variable look is true. If it is true then the lock is synchronized and locked (*synchronized (pause)*) and the lock is made to wait this is blocking a thread (*pauseLock.wait()).* There is a try exception block surrounding the thread block because a thread can unlock itself before time unlocking time is up so an exception is thrown.

*unlock()* method is a setter method it sets the AtomicBoolean variable lock to true in order to initiate a lock.

*unlock()* method is a setter method it sets the variable lock to false, synchronizes the pauseLock and then unlocks the lock and unblock blocked thread.

*getTimeStep()* is an accessor method it returns the number of Time step simulations done by the 4 threads.

*run()* method is setter method that controls the water transfer simulation. The simulation iterates as long as the running variable is set to true by calling the tranferWater(). The for loop is there to make sure the process is continuously as long the condition is not changed. If the running state changes the loop should break out of the current loop, if the pause is true the object should lock by calling the step() method, this will cause a thread block until another thread to calls the .*notifiyAll()* method which unlocks the lock and unblocks the thread. This prevents deadlocks which is a situation where all threads are blocked since they are both dependent on each other both threads will be waiting to access a resource owned by another thread. Therefore, the main thread was made to wait while the other proceeds int his process the water transfer simulation would have been paused. When the waiting time is up the thread is unblocked and proceeds from where it left off. The *transferWater()* method is called and evoked and gets the water transfer up and running again if the state changes. The atomicInteger is incremented to show the end of a water transfer simulation. The window must be repainted at it run to show the change in water location.

1. Flow Class:

This is the main class since it houses the main method which is the entry point where the process of execution starts. A few modifications where made to the class mainly in the *setUPGUI()* method.

A Water object named w was created which takes in the depth of 3 units and Terrain object *landdata* as instance variables. A mouse listener was added to the added to the Flow Panel *fp* to enable the adding of water blocks in grid and the position clicked by the mouse. The water should not be added to the outer points of the grid. Three more buttons were created reset, play and pause and added to the JPanel *b*. All the buttons have an action listener just like the end button, but their actions differ unlike the end button that is supposed to dispose the frame when clicked.

The reset button is supposed to clear all water blocks from the terrain. This process it done by first stopping the *run()* method by calling the *FlowPanel.end()* method that sets running state to false, calling the *Water.removeWater()* method that colors every pixel transparent in the water image and the *FlowPanel.repaint()* that repaints the image component.

The play button is supposed to evoke the start and resume of the water transfer simulation. This is achieved by calling the *FlowPanel.play()* which sets the running state to true in order to start the water transfer simulation. It also calls the *FlowPanel.unpause()* method that sets the pause state to false and unlocks the blocked thread to show that the other thread has finished its execution meaning its run returns. Four threads are created as per assignment requirement and started separately. These should synchronise on each timestep. *Thread.start()* is used to set off a new thread using *run()* as its “main” method to run it in parallel. *Thread.join()* is used to wait for all threads to finish and add together their 4 results for the result.

The pause button is supposed to pause the water transfer simulation. This is achieved by calling the *FlowPanel.pause()* methods that sets the pause state to true.

A JLabel called *timestep* was created. It is upposed to display how many time steps it takes to complete each simulation by the 4 threads.To archive this is calls the FlowPanel.getTimeStep() method from that returns the integer value of the atomicInteger.

Nothing else was changed in the main method.

**Implementations**

Since java does not allow multiple inheritance the more convenient way is to implement the *run()* method from the interface Runnable instead of inheriting from thread this was done by implementing *Runnable* interface in the *FlowPlannel* class.

1. Thread safety

It was ensured in declaring Water.transferWater() method as synchronized to avoid race conditions and by the use of the *unlock()* and *step()* method when called they block and unblock threads in water transfer simulation. The processes are supposed to happen concurrently for the application to function well to allow threads to coordinate. To avoids race conditions which is a bug in a program where the output and or result of the process is unexpectedly and critically depended on the relative sequence of other events, we just wait for all threads to complete their processes. To achieve this there are methods that control the pausing and running of the water these events are not supposed to happen at the same time. So, methods that control the states are called when the pay and pause button are clicked changing the states of the running and pause. If the pause state is on the thread is block if not it is unlocked which causes one thread to wait until another has terminated.

To avoid bad interleaving which is when a wrong result is displayed due to unexpected interleaving of statements in two or more threads, mutual exclusion is implemented. Lock were used to make sure the 4 threads do not try to access the same operation at the same time. If one thread wants to tell something to another thread, it uses *notify()* and *notifyAll()* method of *java.lang.Object.* A call to *notifyAll()* happens when thread changed state of the shared object i.e. When the play button is clicked to resume the water transfer simulation. The *wait()* and *notifyAll()* method must be called from a synchronized context, they should be called in a loop to check the conditions i.e. Whether the play or pause button has been pressed. The *wait()* and *notifyAll()* should be called in a synchronized method or block to avoid a race condition between the *wait()* and *notify()* method. This was archived in the *FlowPanel.unlock()* method.

1. Liveness

A concurrent application's ability to execute in a timely manner is known as its liveness.

1. Deadlocks this is a situation where all threads are blocked. To avoid this I used only one lock for both the running and pausing of the water transfer simulation *pauseLock* and the lock is always acquired in the same order.