5CS022

Distributed and Cloud Systems Programming

Workshop Portfolio

David Pearson

1725412

Contents

Workshop 1 Compiling your first Concurrent Program

Testing a simple threaded program in eclipse. I started by duplicating the two class files provided, though I added comments to the code.

MyThread.java ver1

```
package workshop1;
/**
 * This class describes a thread object
 * @author David Pearson 1725412
 * @param message, this is the message the thread will print
 */
public class MyThread extends Thread{
	// instance variable
	private String message;
	// constructor, takes in a string
	public MyThread(String m) {
		message = m;
	}
	// the run method of the thread object, it will print the message 20 times
	public void run() {
		for (int r = 0; r<20; r++) {
			System.out.println(message);
		}
	}
}
```

TestMyThread.java ver1

```
package workshop1;
/**
 * This class tests the MyThread class
 * @author David Pearson 1725412
 */
public class TestMyThread {
	/** main program logic
	 * Create two my thread objects
	 * set t1 to first thread
	 * set t2 to second thread
	 * start both threads away
	 */
	public static void main(String args[]) {
		MyThread t1,t2;
		t1 = new MyThread("first thread");
		t2 = new MyThread("second thread");
		t1.start();
		t2.start();
	}
}
```

Initially it was not possible to replicate the results shown on the lecture slides, the computer running the program was able to complete the loop, in the run method, in the thread before the second thread had even started. After experimentation I was able to achieve an interleaving of print statements, though this only occurred when the loop counter was increased to a value of over 2000.

INSTRUCTION: Modify the program so that the thread object maintains a counter and print out how many times it prints the message

MyThread.java ver2

```
package workshop1;
/**
 * This class describes a thread object
 * @author David Pearson 1725412
 * @param message, this is the message the thread will print
 * @param counter, this starts at 0 by default and increments
 *                                  for each print statement in the run loop
 */
public class MyThread extends Thread{
        // instance variables
        private String message;
        private int counter;
        // constructor, takes in a string
        public MyThread(String m) {
                message = m;
                counter = 0;
        }
        // the run method of the thread object, it will print the message 'r'
        // number of times, it will also increment the counter
        // after the loop is complete it will print out the counter value
        public void run() {
                for (int r = 0; r<20; r++) {
                        System.out.println(message);
                        counter++;
                }
                System.out.println(message +" has printed "+ counter + " times");
        }
}
```

The modified thread program above meets the requirement, I have printed the results below.

```
second thread
second thread
second thread
second thread
first thread has printed 20 times
second thread has printed 20 times
```

INSTRUCTION: Modify your program to add a third thread, then test the result

TestMyThread.java ver2

```
package workshop1;
/**
 * This class tests the MyThread class
 * @author David Pearson 1725412
 */
public class TestMyThread {
        /** main program logic
         * Create two my thread objects
         * set each thread object up
         * start all threads away
         */
        public static void main(String args[]) {
                MyThread t1,t2,t3;
                t1 = new MyThread("first thread");
                t2 = new MyThread("second thread");
                t3 = new MyThread("third thread");
                t1.start();
                t2.start();
                t3.start();
        }
}
```

The modified main program above meets the requirement, I have printed the results below.

```
second thread
second thread
third thread has printed 20 times
first thread has printed 20 times
second thread has printed 20 times
```

Workshop 1 A Multithread Program

I started by recreating the two class files provided from the lecture slides, adding comments to the code to aid my understanding.

Worker.java ver1

```
package workshop1part2;
/**
 * This class describes a worker object
 * @author David Pearson 1725412
 * because it implements runnable it allows there to be a run method
 * the other method pause sets the Thread to sleep for a random amount
 * of time.
 */
public class Worker implements Runnable{

	public void pause() {
		int delay = (int)(Math.random()*500);
		try {
			Thread.sleep(delay);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
	public void run() {
		for (int i=0; i<5; i++) {
			pause();
			System.out.println(Thread.currentThread().getName());
		}
	}
}
```

RunWorkersDemo.java ver1

```
package workshop1part2;
/**
 * This class tests the Worker class
 * @author David Pearson 1725412
 */
public class RunWorkersDemo {
	/** main program logic
	 * Declare 3 worker objects
	 * Declare 3 thread objects
	 * Create a a start time variable
	 * Create 3 worker objects
	 * Create 3 Thread objects passing the worker objects
	 * Start all thread onjects
	 * attempt to rejoin all threads
	 * print out time taken
	 */
	public static void main(String[] args) {
		Worker w1,w2,w3;
		Thread t1,t2,t3;
		long start = System.nanoTime();

		w1 = new Worker();
		w2 = new Worker();
		w3 = new Worker();

		t1 = new Thread(w1);
		t2 = new Thread(w2);
		t3 = new Thread(w3);

		t1.start();
		t2.start();
		t3.start();

		try {
			t1.join();
			t2.join();
			t3.join();
		} catch (Exception e) {
			System.out.println("problem waiting for thread tofinish");
		}
		long duration = System.nanoTime() -start;
		System.out.printf("running time = %d nano seconds or %fs\n", duration,duration*1e-9);
	}
}
```

I was able to recreate the results shown in the lecture slides, showing that the threads are run in parallel.

```
Thread-2
Thread-0
Thread-1
Thread-2
Thread-0
Thread-2
Thread-1
Thread-1
Thread-2
Thread-0
Thread-1
Thread-1
Thread-0
Thread-2
Thread-0
running time = 1387524821 nano seconds or 1.387525s
```

INSTRUCTION: Modify your program to add a forth thread, then test the results.

RunWorkersDemo.java ver2

```java
package workshop1part2;
/**
 * This class tests the Worker class
 * @author David Pearson 1725412
 */
public class RunWorkersDemo {
        /** main program logic
         * Declare 4 worker objects
         * Declare 4 thread objects
         * Create a a start time variable
         * Create 4 worker objects
         * Create 4 Thread objects passing the worker objects
         * Start all thread onjects
         * attempt to rejoin all threads
         * print out time taken
         */
        public static void main(String[] args) {
                Worker w1,w2,w3,w4;
                Thread t1,t2,t3,t4;
                long start = System.nanoTime();

                w1 = new Worker();
                w2 = new Worker();
                w3 = new Worker();
                w4 = new Worker();

                t1 = new Thread(w1);
                t2 = new Thread(w2);
                t3 = new Thread(w3);
                t4 = new Thread(w4);

                t1.start();
                t2.start();
                t3.start();
                t4.start();

                try {
                        t1.join();
                        t2.join();
                        t3.join();
                        t4.join();
                } catch (Exception e) {
                        System.out.println("problem waiting for thread tofinish");
                }
                long duration = System.nanoTime() -start;
                System.out.printf("running time = %d nano seconds or %fs\n", duration,duration*1e-9);
        }
}
```

Adding a forth thread increased the overall runtime of the program. It also again showed parallel execution.

```
Thread-1
Thread-0
Thread-3
Thread-3
running time = 1655585482 nano seconds or 1.655585s
```

Workshop 2 PennyAdder Program

I successfully installed the three java class files as described in the workshop document.

Account.java ver1

```
package workshop2;

public class Account {
    private int balance = 0;
    public int getBalance() {return balance;}
    public void setBalance(int b) { balance = b;}
}
```

Bank.java ver1

```
package workshop2;
public class Bank {

    private static Account account = new Account();
    private static Thread[] threads  = new Thread[100];

        public static void main(String[] args) {
                boolean done = false;
                for (int i = 0; i < 100; i++){
                    threads[i] = new Thread(new PennyAdder(account), "thread-"+i);
                    threads[i].start();
                }
                // wait for all threads to finish
                for (int i = 0; i < 100; i++){
                    try {
                                threads[i].join();
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
                System.out.println("The balance is " + account.getBalance());
        }
}
```

PennyAdder.java ver1

```
package workshop2;

public class PennyAdder implements Runnable {
    private Account account;
    public PennyAdder(Account a){
        account = a;
    }
    public void addPenny() {
        System.out.println("addPenny called from " + Thread.currentThread().getName());
        // get the current balance of the account
        int balance = account.getBalance();
        // simulate a delay in calculating the new balance
        try {
                        Thread.sleep(1);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        // add a penny
        balance = balance + 1;
        // update the balance
        account.setBalance(balance);
        System.out.println("penny added by thread " + Thread.currentThread().getName());
    }
    public void run(){
        addPenny();
    }
}
```

Running the program several times, it was possible to replicate the results shown in the lecture slides.

```
penny added by thread thread-93
penny added by thread thread-98
penny added by thread thread-96
penny added by thread thread-99
penny added by thread thread-97
The balance is 9
```

```
penny added by thread thread-92
penny added by thread thread-90
penny added by thread thread-93
penny added by thread thread-89
penny added by thread thread-94
The balance is 11
```

INSTRUCTION: Modify the PennyAdder class to include synchronisation (FixedPennyAdder). Runthe code again to confirm the results are now correct.

PennyAdder.java ver2

```
package workshop2;
public class PennyAdder implements Runnable {
    private Account account;
    public PennyAdder(Account a){
        account = a;
    }
    public void addPenny() {
         synchronized(account) {
        System.out.println("addPenny called from " + Thread.currentThread().getName());
        // get the current balance of the account
        int balance = account.getBalance();
        // simulate a delay in calculating the new balance
        try {
                        Thread.sleep(1);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        // add a penny
        balance = balance + 1;
        // update the balance
        account.setBalance(balance);
        System.out.println("penny added by thread " + Thread.currentThread().getName());
         }
    }
    public void run(){
        addPenny();
    }
}
```

I successfully modified the program by adding synchronisation to the account object. This corrected the race condition errors that were occurring. I was able to achieve the desired results.

It is also clear as seen in these results that the order of the threads is random.

```
addPenny called from thread-16
penny added by thread thread-16
addPenny called from thread-15
penny added by thread thread-15
addPenny called from thread-14
penny added by thread thread-14
The balance is 100
```

```
addPenny called from thread-21
penny added by thread thread-21
addPenny called from thread-20
penny added by thread thread-20
addPenny called from thread-19
penny added by thread thread-19
The balance is 100
```

Workshop 2 PennyAdder2

I successfully installed the three java class files as described in the workshop document.

Accoun2.java ver1

```
package workshop2part2;

public class Account2 {
    private int balance = 0;
    public int getBalance() {return balance;}
    public void setBalance(int b) { balance = b;}
}
```

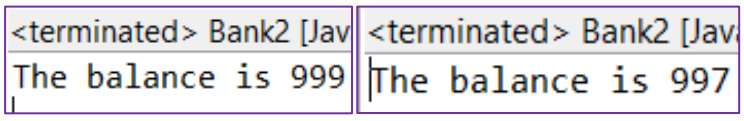
Bank2.java ver1

```
package workshop2part2;
public class Bank2 {
    private static Account2 account = new Account2();
    private static Thread[] threads  = new Thread[1000];

        public static void main(String[] args) {
                boolean done = false;

                for (int i = 0; i < 1000; i++){
                    threads[i] = new Thread(new PennyAdder2(account), "thread-"+i);
                    threads[i].start();
                }

                // wait for all threads to finish
                for (int i = 0; i < 1000; i++){
                    try {
                                threads[i].join();
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                }
                System.out.println("The balance is " + account.getBalance());
        }
}
```

PennyAdder2.java ver1

```
package workshop2part2;

public class PennyAdder2 implements Runnable {
    private Account2 account;
    public PennyAdder2(Account2 a){
        account = a;
    }
    public void addPenny() {
        account.setBalance(account.getBalance() + 1);
    }
    public void run(){
        addPenny();
    }
}
```

After adding the files into my IDE and running the main method I was able to reproduce the output described in the lecture.



INSTRUCTION: Modify PennyAdder2 to use AtomicInteger. Run the code again to confirm the results are now correct.

Account2.java ver2

```
package workshop2part2;

import java.util.concurrent.atomic.AtomicInteger;

public class Account2 {
        private AtomicInteger balance;
    public Account2 (AtomicInteger ai) { balance = ai;}
    public AtomicInteger getBalance() {return balance;}
    public void setBalance(int b) { balance.set(b);}
}
```

Bank2.java ver2

```
package workshop2part2;
import java.util.concurrent.atomic.AtomicInteger;
public class Bank2 {
    private static Account2 account = new Account2(new AtomicInteger(0));
    private static Thread[] threads  = new Thread[1000];
        public static void main(String[] args) {
                for (int i = 0; i < 1000; i++){
                    threads[i] = new Thread(new PennyAdder2(account), "thread-"+i);
                    threads[i].start();
                }
                // wait for all threads to finish
                for (int i = 0; i < 1000; i++){
                    try {
                                threads[i].join();
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
                System.out.println("The balance is " + account.getBalance());
        }
}
```

PennyAdder2.java ver2

```
package workshop2part2;

public class PennyAdder2 implements Runnable {
    private Account2 account;
    public PennyAdder2(Account2 a){
        account = a;
    }
    public void addPenny() {
         System.out.println("addPenny called from " +Thread.currentThread().getName());
         account.getBalance().incrementAndGet();
         System.out.println("penny added by thread " +Thread.currentThread().getName());
    }
    public void run(){
        addPenny();
    }
}
```

After modifying the files by changing the balance variable of type int to AtomicInteger this removed the problems with concurrency. Though as before by adding in the print statements to show when each thread is acting, again we can see they are not in sequential order.

```
addPenny called from thread-993
penny added by thread thread-993
addPenny called from thread-996
penny added by thread thread-996
addPenny called from thread-997
penny added by thread thread-997
addPenny called from thread-998
penny added by thread thread-998
addPenny called from thread-999
penny added by thread thread-999
The balance is 1000
```

```
penny added by thread thread-995
addPenny called from thread-996
penny added by thread thread-996
addPenny called from thread-997
penny added by thread thread-997
addPenny called from thread-998
penny added by thread thread-998
addPenny called from thread-999
penny added by thread thread-999
The balance is 1000
```

Workshop 2 Producer – Consumer

INSTRUCTION: Write a concurrent program to count the number people entering a garden through either of two gates (North or South). Management would like to know how many people are in the garden at any time. There should be two concurrent threads that simulate the periodic arrival of a visitor to the garden every second (hint: use sleep) and a shared counter object for the number of people in the garden. You can close the gates after each has let about 20 people to enter.

To meet the specification above three classes were created. A Garden class, a Gate class and a Main class which is where the programs main method was called.

Garden.java ver1

```
package workshop2part3;
public class Garden {
	private int capacity = 0;
	private boolean usable = false;

	public synchronized int getCapacity() {
		while(!usable) {
			try {
				wait();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
		usable = false;
		notifyAll();
		return capacity;
	}
	public synchronized void addPeson() {
		while(usable) {
			try {
				wait();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
		usable = true;
		capacity++;
		notifyAll();
	}
}
```

Gate.java ver1

```
package workshop2part3;
public class Gate implements Runnable{

	Garden g1;
	String GateID;
	int gatecount;
	public Gate(Garden g1, String id) {
		this.g1 = g1;
		GateID = id;
		gatecount=0;
	}
	public void run() {
			while (gatecount< 20) {
				g1.addPeson();
				gatecount++;
				System.out.println("Someone just entered the garden through "+GateID
				+", the garden has "+g1.getCapacity()+" people in it");
				try {
					Thread.sleep((long)(Math.random()*1000));
				}catch (InterruptedException e) {
					e.printStackTrace();
				}
			}
	}
}
```

Main.java ver1

```
package workshop2part3;
public class Main {
        public static void main(String[] args) {
                Garden garden = new Garden();
                Gate GateOneIn = new Gate(garden,"North Gate");
                Gate GateTwoIn = new Gate(garden,"South Gate");
                new Thread(GateOneIn).start();
                new Thread(GateTwoIn).start();
        }
}
```

When running the program and using two gates I was able to produce the following results

```
Someone just entered the garden through South Gate, the garden has 24 people in it
Someone just entered the garden through North Gate, the garden has 25 people in it
Someone just entered the garden through North Gate, the garden has 26 people in it
Someone just entered the garden through South Gate, the garden has 27 people in it
Someone just entered the garden through North Gate, the garden has 28 people in it
Someone just entered the garden through North Gate, the garden has 29 people in it
Someone just entered the garden through South Gate, the garden has 30 people in it
Someone just entered the garden through North Gate, the garden has 31 people in it
Someone just entered the garden through South Gate, the garden has 32 people in it
Someone just entered the garden through North Gate, the garden has 33 people in it
Someone just entered the garden through North Gate, the garden has 34 people in it
Someone just entered the garden through North Gate, the garden has 35 people in it
Someone just entered the garden through North Gate, the garden has 36 people in it
Someone just entered the garden through South Gate, the garden has 37 people in it
Someone just entered the garden through South Gate, the garden has 38 people in it
Someone just entered the garden through South Gate, the garden has 39 people in it
Someone just entered the garden through South Gate, the garden has 40 people in it
```

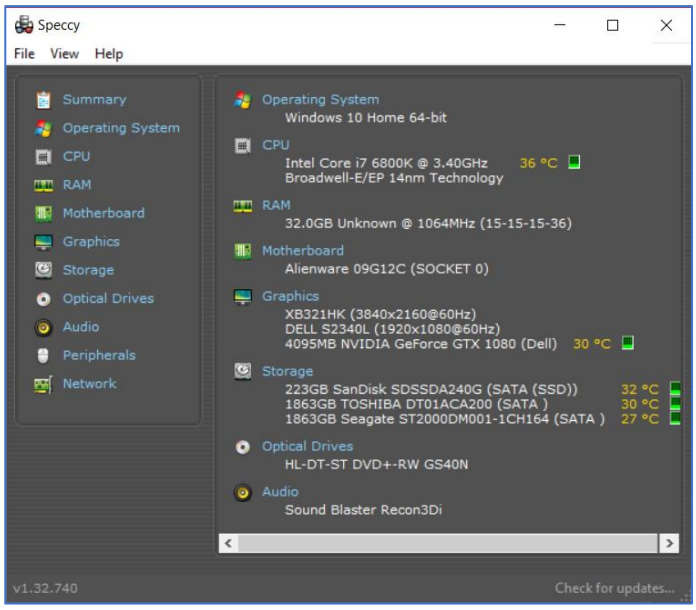
```
Someone just entered the garden through South Gate, the garden has 24 people in it
Someone just entered the garden through North Gate, the garden has 25 people in it
Someone just entered the garden through North Gate, the garden has 26 people in it
Someone just entered the garden through South Gate, the garden has 27 people in it
Someone just entered the garden through North Gate, the garden has 28 people in it
Someone just entered the garden through South Gate, the garden has 29 people in it
Someone just entered the garden through North Gate, the garden has 30 people in it
Someone just entered the garden through North Gate, the garden has 31 people in it
Someone just entered the garden through South Gate, the garden has 32 people in it
Someone just entered the garden through North Gate, the garden has 33 people in it
Someone just entered the garden through South Gate, the garden has 34 people in it
Someone just entered the garden through North Gate, the garden has 35 people in it
Someone just entered the garden through North Gate, the garden has 36 people in it
Someone just entered the garden through North Gate, the garden has 37 people in it
Someone just entered the garden through South Gate, the garden has 38 people in it
Someone just entered the garden through North Gate, the garden has 39 people in it
Someone just entered the garden through North Gate, the garden has 40 people in it
```

This was achieved by placing synchronisation on the methods called by my gate class within the garden class, rather than have that on the capacity variable itself. A lock was also used by way of a Boolean variable, this in combination with the sleep/notify all methods ensured concurrency.

Workshop 2 Timing exercise

How many threads will the JVM allow you to create?

Before starting to test this question, I hold a firm belief that the answer will vary from system to system, for example I will be conducting my tests on my home computer which has the following specifications.



To carry out the tests I made use of the code from task 1 of this workshop. Modifying the two for loops as well as the Thread array size, within Bank.java.

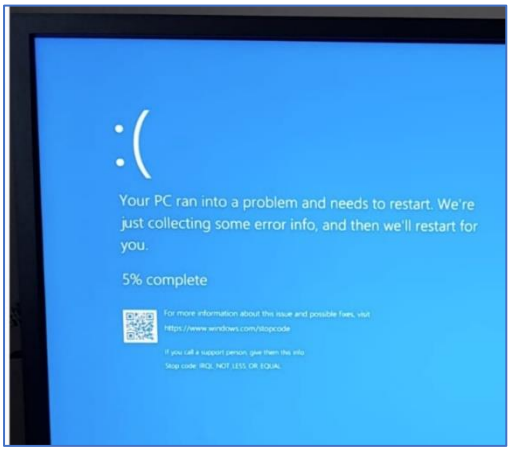
Unfortunately, my first test of setting the count to 1 million resulted in an entire system crash as seen below.

I was able to successfully spawn 10,000 threads.

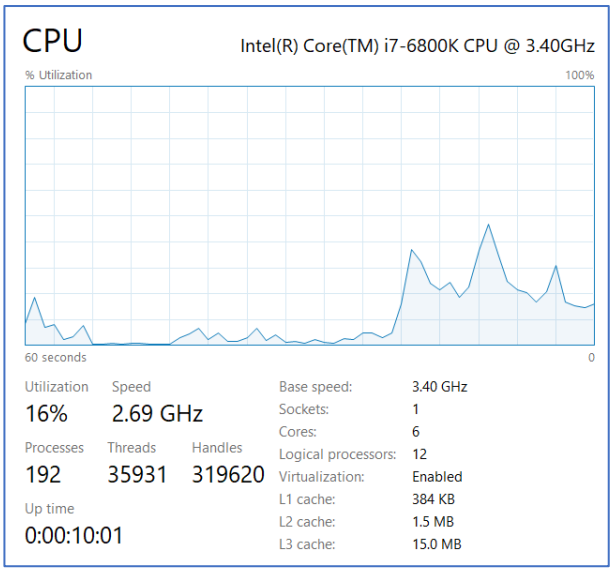
I was able to successfully spawn 50,000 threads.

When I attempted 100,000 threads, again my system crashed. Though whilst monitoring my system whilst the teat was being carried out, there appeared to be no obvious strain on the system. So, I attempted to run for 100,000 again and for the third time in my experiments I had generated the result of a blue screen of death. You can see where I started the java program off clearly in the monitor.

The other element of this question asks how long it takes to create the threads, in the tests that I carried out it was clear that the more threads that were created, the more time it took.



This clearly implies that the number of threads used in the creation of concurrent programs will influence both the hardware requirements as well as the efficiency of the program.



Workshop 5 Concurrent and Cloud Intro

INSTRUCTION Change the textfile to pickup the LongFellowPoem.txt

As the program stood it loaded the filename specified within the constructor of the Server object. My chosen solution to this involved asking the user to also specify a filename they wanted to load at the same time they passed in the port number. This was achieved by altering several methods, detailed below

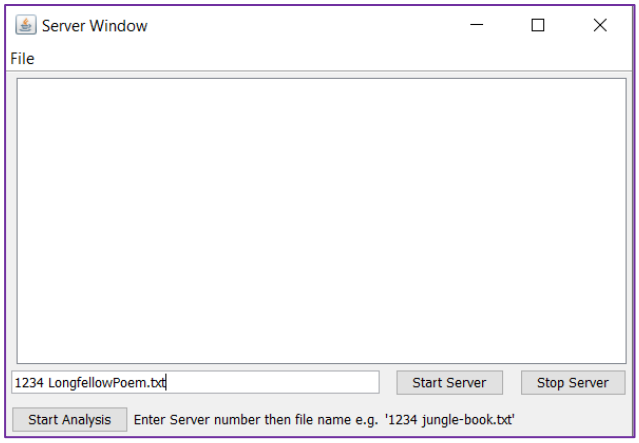
ServerWindow.java – File read change

```
//Create a JTextField that is going to be used to enter commands
txtMessage = new JTextField();
//Adding a key listener
txtMessage.addKeyListener(new KeyAdapter() {
        /**
         * This action listener checks for the enter key being pressed.
         * if the enter key is pressed it slpits the text in the
         * JTextField into an integer and a string
         * the int will be the port and the string the filename
         * It then creates a Server object passing in the port entered
         * and the filename entered
         * it then sets the server objects Text area to the text area
         * of the Server Window
         * Finally it clears the txtMessage box
         *
         */
        public void keyPressed(KeyEvent e) {
                if (e.getKeyCode() == KeyEvent.VK_ENTER) {
                        String textbox = txtMessage.getText();
                        int port = Integer.parseInt(textbox.substring(0, 4));
                        String filename = textbox.substring(5);
                        System.out.println(port);
                        System.out.println(filename);
                        server = new Server(port,filename);
                        server.output = history;
                        txtMessage.setText("");
                }
        }
});
```

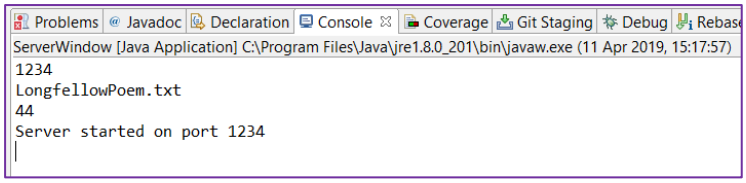
Server.java – File read change

```
/**
 * Constructor for a Server Object
 * @param port 4 digit number used to createthe socket
 * @param file filename to be analysed
 * First tries create a datagramsocket and assign it to the
 * Socket
 *
 * Creates a buffer reader called in and reads in the file
 * passed in
 * Create a String called str
 * create an arraylist to contain strings called SetList
 * Load the file into the arraylist
 * print the number of lines in the file
 * Create a thread called server and set it away
 *
 */
public Server(int port, String filename) {
        this.port = port;
        this.filename = filename;
        try {
                socket = new DatagramSocket(port);
        } catch (SocketException e) {
                e.printStackTrace();
                return;
        }

        try {
                in = new BufferedReader(new FileReader(filename));
        } catch (FileNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
```

Below is a screen showing the loading of the Longfellow poem and starting a server on port 1234.



When the "Start Server" button is clicked, confirmation can be seen in the console;



INSTRUCTION Search for the instances of the letter 'e' in this 36 line poem

Initially the functionality within the program to complete the alphabet count contained a number of errors. I was able to resolve these over a period of weeks and made use of GIT to track my progress, as each change was made.

Not wanting to reinvent the wheel, I simply choose to use the result of the alphabet count, as this method views each character in turn anyway. It would have been possible to modify the method with a simple Boolean evaluation along the lines of, does character 'x' == e, if so increment a counter.

The number of instances of the letter 'e' in the poem was = 104

INSTRUCTION Change the number of clients to see how the over-all execution time is affected

What appears to be the optimal number of clients?

Approaching this task objectively, when the "start analysis" button is pressed, it first compiles a list of attached clients then sends them each a line. As the poem only has 44 lines, having more than 44 clients would mean that client 45 would perform no analysis at all.

This gives 44 as the maximum number of clients, logically 1 is the lower boundary to test.

Starting by first performing the operation using these upper and lower boundaries, plus and minus one, the results are shown below;

Number of Clients	First run Time in ms	Second run Time in ms	Third run Time in ms	Average Time in ms
1	63	61	83	69
2	49	46	50	48
43	46	53	41	47
44	47	56	45	49

Seeing the results thus far, it is apparent that the optimal number of clients would be 43. However, I wanted to test some further client numbers around the mid-range also, these results can be seen below;

Number of Clients	First run Time in ms	Second run Time in ms	Third run Time in ms	Average Time in ms
10	40	38	51	46
20	40	41	42	41
30	38	34	38	37
40	38	43	40	40

I hold a strong belief that the optimal number of clients is far from a static number, a great many factors can influence this variable, including things such as the scheduler, other software being run on the system and the system itself, to name but a tiny portion.

Appendix

Workshop 1 Document

5CS022 Week 01 Activities

For this workshop you will be ensuring you can use eclipse and testing some simple threaded program.

Note: *if you copy'n'paste text from the lecture notes (or this document) you may have to edit the quotes ("), semicolons (;) and mathematical operators (+-*/), as the copy operation often substitutes them for Unicode characters.*

Setting up a module workspace and Environment

It is recommended create a module folder called 5CS022 for your work and within it a subfolder, Week01, for this week's work. Then Install Eclipse from Apps anywhere and launch it.

1 - Compiling your first Concurrent program

Create a new project call it myThread and within the project create two classes call them MyThread and TestMyThread.

```
class MyThread extends Thread {
   private String message;

   public MyThread(String m) {
      message = m;
   }

   public void run() {
      for (int r=0; r<20; r++)
         System.out.println(message);
   }
}
```

```
public class TestMyThread {

   public static void main(String[] args) {
      MyThread t1,t2;
      t1=new MyThread("first thread");
      t2=new MyThread("second thread");
      t1.start();
      t2.start();
   }
}
```

Use the code from this morning's lecture for MyThread and TestMyThread.

Build and run the program to confirm you can produce a similar output to that described in the lecture notes.

Modify the program so that the thread object maintains a counter and print out how many times it prints the message.

Modify your program to add a third thread, then test the result

2 - A Multithread Program

Create a new project call it workersExample. Copy the code from the RunWorkersDemo program given in the lecture and incorporate into the workersExample project.

Build and run the program to confirm you can produce a similar output to that described in the lecture notes.

- Modify your program to add a fourth thread, then test the results.

5CS022 Workshop 02 Activities

Concurrent programs

It is recommended create a subfolder called Week02, for this week's work. Then Install Eclipse from Apps anywhere and launch it.

1 - PennyAdder program

Create a new project call it pennyAdder. Download and incorporate the code from the pennyAdder zip on Canvas.

Build and run the program to confirm you can produce a similar output to that described in the lecture notes.

- Modify the PennyAdder class to include synchronisation (FixedPennyAdder). Run the code again to confirm the results are now correct.

2 - PennyAdder2

Create a new project call it pennyAdder2. Download and incorporate the code from the pennyAdder2 zip on Canvas.

Build and run the program to confirm you can produce a similar output to that described in the lecture notes.

- Modify PennyAdder2 to use AtomicInteger. Run the code again to confirm the results are now correct.

3 - Producer-Consumer

Write a concurrent program to count the number people entering a garden through either of two gates (North or South). Management would like to know how many people are in the garden at any time.

There should be two concurrent threads that simulate the periodic arrival of a visitor to the garden every second (hint: use sleep) and a shared counter object for the number of people in the garden. You can close the gates after each has let about 20 people to enter.

4 - Timing exercise

1. How many threads will the JVM allow you to create? How long does it take to create them? Write a program investigate this on your machine. What does this imply for those creating concurrent programs

Workshop 5 screen grab