CST-239 Activity 6 Guide

Contents

Part 1: Creating a Thread

Overview

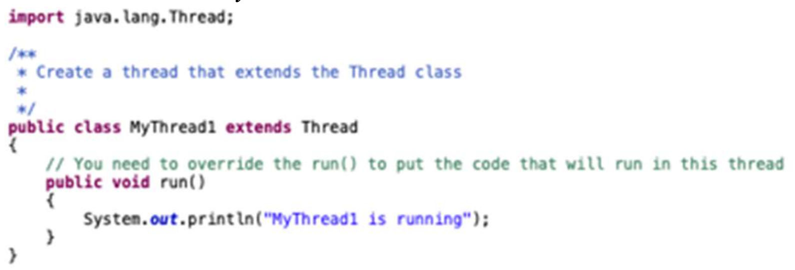
Goal and Directions:

In this activity you learn how to create threads using the Java Thread classes as well how to make a multi-threaded application thread safe.

Execution

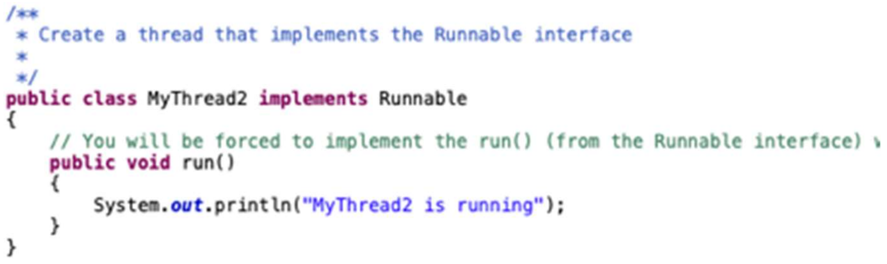
Part 1 – Simple Threads

1. Create a new Java Project named *topic6-1a*.
2. Create a new class named *MyThread1* class in the *app* package. Implement the class as follows:
    a. Extend from java.lang.Thread class.
    b. Implement the void run() method by simply printing a message to the console of the name of the *MyThread1* class.



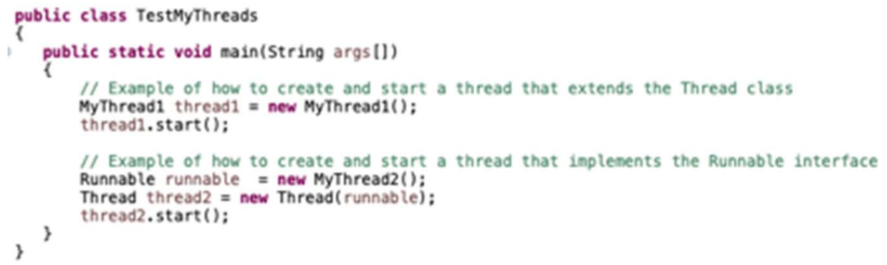
```
import java.lang.Thread;

/**
 * Create a thread that extends the Thread class
 *
 */
public class MyThread1 extends Thread
{
    // You need to override the run() to put the code that will run in this thread
    public void run()
    {
        System.out.println("MyThread1 is running");
    }
}
```

3. Create a new class named *MyThread2* class in the *app* package. Implement the class as follows:
    a. Implement the Runnable interface.
    b. Implement the void run() method by simply printing a message to the console of the name of the *MyThread2* class.



```java
/**
 * Create a thread that implements the Runnable interface
 *
 */
public class MyThread2 implements Runnable
{
    // You will be forced to implement the run() (from the Runnable interface)
    public void run()
    {
        System.out.println("MyThread2 is running");
    }
}
```

4. Create a new class named *TestMyThreads* class in the *app* package with a *main*(). Implement the logic in the *main()* as follows:
    a. Create an instance of the *MyThread1* class and call the *start()* method on the class.
    b. Create an instance of the *MyThread2* class and save this in a local variable of type Runnable. Create an instance of a Thread class passing the variable as an argument to constructor argument and call the *start()* method on the class.

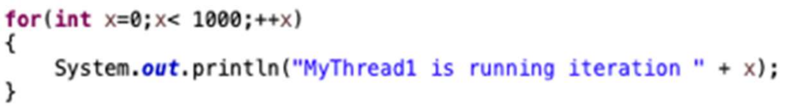


```java
public class TestMyThreads
{
    public static void main(String args[])
    {
        // Example of how to create and start a thread that extends the Thread class
        MyThread1 thread1 = new MyThread1();
        thread1.start();

        // Example of how to create and start a thread that implements the Runnable interface
        Runnable runnable  = new MyThread2();
        Thread thread2 = new Thread(runnable);
        thread2.start();
    }
}
```

5. Run the *TestMyThreads* application.
6. Take a screen shot of the console output.
7. Remember, the threads are sharing the CPU time when running the application code, the code in MyThread1, and MyThread2. Provide a brief (3-4 sentences) description how and why the output got displayed.

<u>Part 2 – More Complex Threads</u>

1. Create a new project named *topic6-1b*. Copy all the code from *topic6-1a* to the new project. Run the test cases in the new *topic6-1b* project to ensure the new project is working properly.
2. Modify the *MyThread1* class as follows:
    a. Add a for loop in the run() method for up to 100 iterations and that for each iteration:
        i. Prints a unique message to the console (like a string with the name of the class name concatenated with the value of for loop iteration variable).

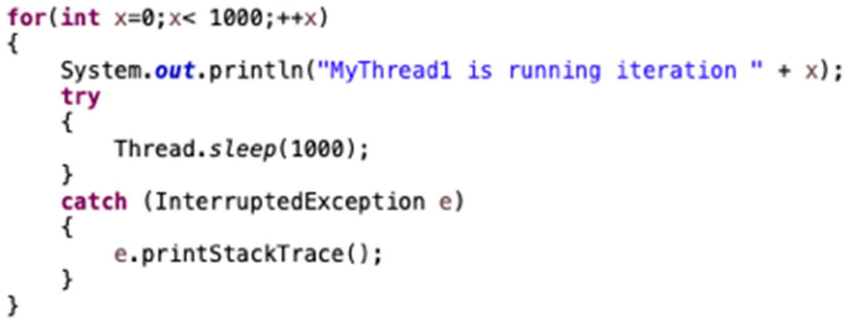


```java
for(int x=0;x< 1000;++x)
{
    System.out.println("MyThread1 is running iteration " + x);
}
```

3. Modify the *MyThread2* class as follows.

a. Add a for loop in the run() method for up to 100 iterations and that for each iteration:
    i. Prints a unique message to the console (like a string with the name of the class name concatenated with the value of for loop iteration variable). See the above for an example code snippet.

4. Run the *TestMyThreads* application.
5. Take a screen shot of the console output.
6. Remember, the threads are sharing the CPU time when running the application code, the code in MyThread1, and MyThread2. Why does it appear that all loop iterations of MyThread1 got executed and then all loop iterations of MyThread2 got executed? Did it really run MyThread1 and then run MyThread2 sequentially as the output showed? Provide a brief (3-4 sentences) description how and why the output got displayed.
7. Add thread sleep code (of Thread.sleep(1000)) in MyThread1 after the print statement within the for loop. Add thread sleep code (of Thread.sleep(500)) in MyThread2 after the print statement within the for loop.

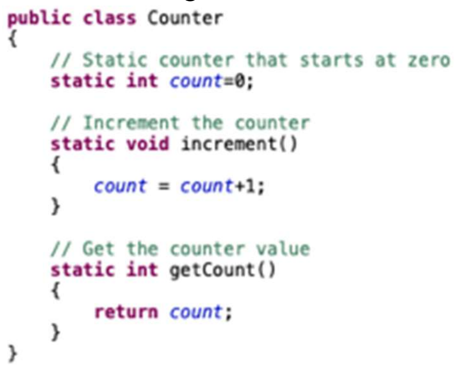


```
for(int x=0;x< 1000;++x)
{
    System.out.println("MyThread1 is running iteration " + x);
    try
    {
        Thread.sleep(1000);
    }
    catch (InterruptedException e)
    {
        e.printStackTrace();
    }
}
```

8. Run the *TestMyThreads* application. Click the Stop icon in the console to stop the execute of the application.
9. Take a screen shot of the console output.
10. Remember, the threads are sharing the CPU time when running the application code, the code in MyThread1, and MyThread2. How did putting the thread to sleep for different amounts of time cause the output to be changed? Provide a brief (3-4 sentences) description how and why the output got displayed.

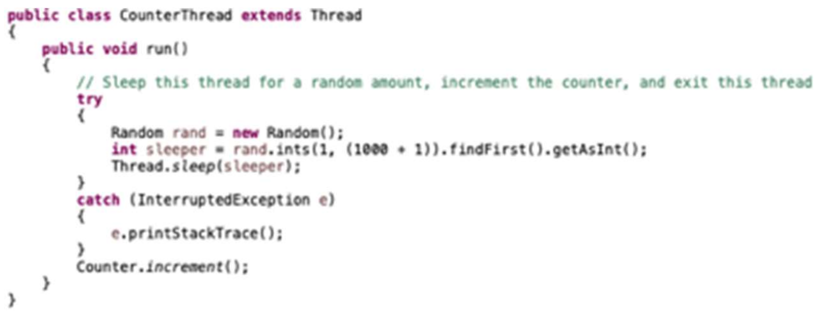
Part 3 – Threads with a Concurrency Problem and Fixing the Problem

1. Create a new Java Project named *topic6-1c*.
2. Create a new class *Counter* in the *app* package that contains a static counter value with static methods to increment the counter value and get the counter value.



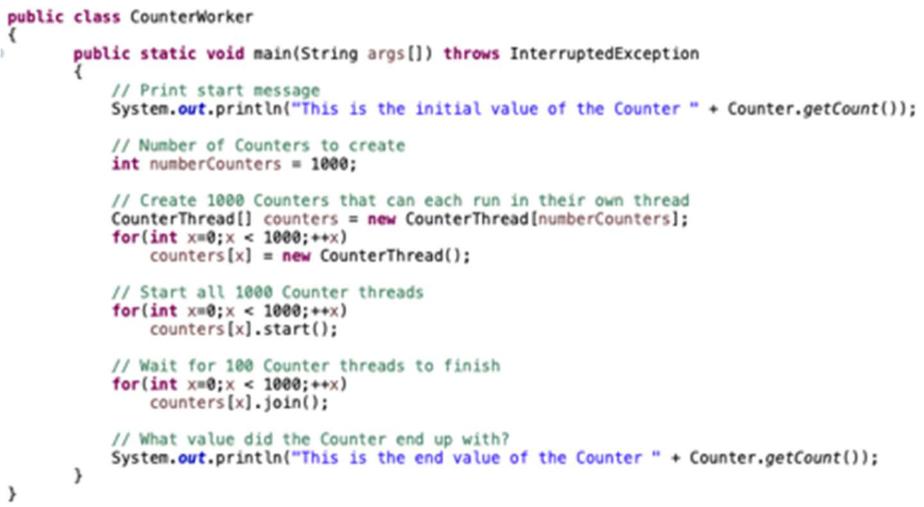
```
public class Counter
{
    // Static counter that starts at zero
    static int count=0;

    // Increment the counter
    static void increment()
    {
        count = count+1;
    }

    // Get the counter value
    static int getCount()
    {
        return count;
    }
}
```

3. Create a new class *CounterThread* in the *app* package that extends from the Thread class and whose *run()* method generated a random number between 1 and 1000, uses the random number to make the sleep for that amount of milliseconds, and then calls the static *increment()* method on the *Counter*.



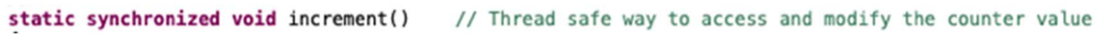
```
public class CounterThread extends Thread
{
    public void run()
    {
        // Sleep this thread for a random amount, increment the counter, and exit this thread
        try
        {
            Random rand = new Random();
            int sleeper = rand.ints(1, (1000 + 1)).findFirst().getAsInt();
            Thread.sleep(sleeper);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
        Counter.increment();
    }
}
```

4. Create a new class *CounterWorker* in the *app* package that has a main() method, which prints to the console the value of the *Counter* by calling the static *getCount()* method, creates and initializes an array contains 1000 instances of the *CounterThread* classes, starts the 1000 instances of the *CounterThread* threads, waits for all the 1000 instances of the *CounterThread* threads to finish, and finally prints out the end value of the *Counter* by calling the static *getCount()* method.



```
public class CounterWorker
{
        public static void main(String args[]) throws InterruptedException
        {
            // Print start message
            System.out.println("This is the initial value of the Counter " + Counter.getCount());

            // Number of Counters to create
            int numberCounters = 1000;

            // Create 1000 Counters that can each run in their own thread
            CounterThread[] counters = new CounterThread[numberCounters];
            for(int x=0;x < 1000;++x)
                counters[x] = new CounterThread();

            // Start all 1000 Counter threads
            for(int x=0;x < 1000;++x)
                counters[x].start();

            // Wait for 100 Counter threads to finish
            for(int x=0;x < 1000;++x)
                counters[x].join();

            // What value did the Counter end up with?
            System.out.println("This is the end value of the Counter " + Counter.getCount());
        }
}
```

5. Run the *CounterWorker* application.
6. Take a screen shot of the console output.
7. Remember, the threads are sharing the CPU time when running the application code, the code is running 1000 threads each randomly incrementing the counter by 1. The end result if execution was correct would have expected the final counter value to be 1000 (each of the 1000 threads each increments the counter by 1). Why did the counter not end up at a value of 1000 as expected? Provide a brief (3-4 sentences) description how and why the output got displayed.
8. Let's fix the concurrency problem in the multi-threaded code.
9. Make the access to incrementing the counter thread safe by making the method synchronized using the Java synchronized keyword.



```
static synchronized void increment()    // Thread safe way to access and modify the counter value
```

10. Run the *CounterWorker* application.
11. Take a screen shot of the console output.
12. The end result if execution is now was correct with the final counter value at 1000 (each of the 1000 threads each increments the counter by 1). How did the simple code update fix the problem? Provide a brief (3-4 sentences) description how and why the output got displayed.

<u>Deliverables</u>:

The following needs to be submitted as this part of the Activity:

a. Theory of operation write ups.
b. All screenshots of application in operation.
c. ZIP file of the code in the project folder. Include the JavaDoc generated for the project.

GRAND CANYON UNIVERSITY

Part 2: Creating a Client and Server Networking Application

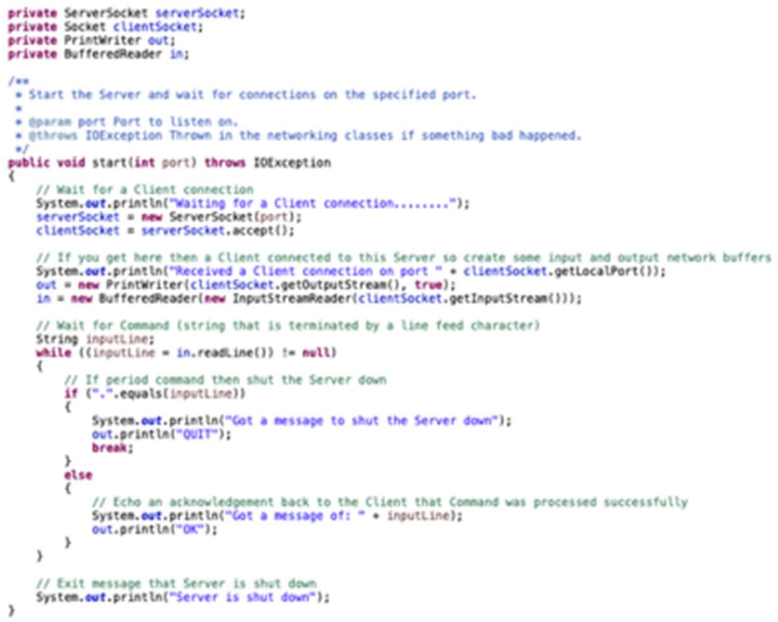
Overview

Goal and Directions:

You will also learn how to use the Java Networking classes to create a client server application that can send information back and forth, and then finally how to create a multi-threaded version of the client server application.

Execution

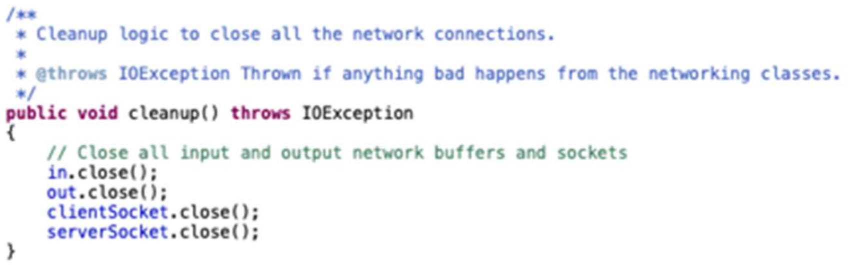
1. Create a new Java Project named *topic6-2*.
2. Create a new class *Server* in the *app* package with a *main()* method to receive and process incoming messages:
    a. Create a method named *start()* method that takes a port as parameter, returns void, and throws IOException.
        i. Create a *ServerSocket* to connect to the specified port.
        ii. Wait for a connection from the Client by calling *accept()* on the socket.
        iii. Create a *PrintWriter* for sending text over the socket to the Client.
        iv. Create a *BufferReader* for receiving text over the socket from the Client.
        v. Loop forever printing all incoming message from the Client and checking for a message of dot (".") which is a message to quit. If the quit message is not received send an OK ("OK") message back to the Client. If the message is quit message then send quite ("QUIT") message back to the Client and break out the loop to return to the *main()* method.



```
private ServerSocket serverSocket;
private Socket clientSocket;
private PrintWriter out;
private BufferedReader in;

/**
 * Start the Server and wait for connections on the specified port.
 *
 * @param port Port to listen on.
 * @throws IOException Thrown in the networking classes if something bad happened.
 */
public void start(int port) throws IOException
{
    // Wait for a Client connection
    System.out.println("Waiting for a Client connection........");
    serverSocket = new ServerSocket(port);
    clientSocket = serverSocket.accept();

    // If you get here then a Client connected to this Server so create some input and output network buffers
    System.out.println("Received a Client connection on port " + clientSocket.getLocalPort());
    out = new PrintWriter(clientSocket.getOutputStream(), true);
    in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

    // Wait for Command (string that is terminated by a line feed character)
    String inputLine;
    while ((inputLine = in.readLine()) != null)
    {
        // If period command then shut the Server down
        if (".".equals(inputLine))
        {
            System.out.println("Got a message to shut the Server down");
            out.println("QUIT");
            break;
        }
        else
        {
            // Echo an acknowledgement back to the Client that Command was processed successfully
            System.out.println("Got a message of: " + inputLine);
            out.println("OK");
        }
    }

    // Exit message that Server is shut down
    System.out.println("Server is shut down");
}
```

b. Create a method named *cleanup()* method that takes no parameters, returns void, and throws IOException.

  i. Close all network buffers and sockets.



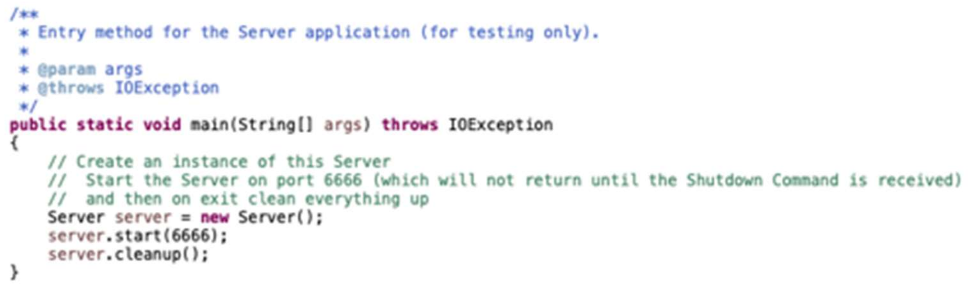
```
/**
 * Cleanup logic to close all the network connections.
 *
 * @throws IOException Thrown if anything bad happens from the networking classes.
 */
public void cleanup() throws IOException
{
    // Close all input and output network buffers and sockets
    in.close();
    out.close();
    clientSocket.close();
    serverSocket.close();
}
```

c. Implement the *main()* method.

  i. Create an instance of the *Server* class.

  ii. Call the *start()* method on the Server (this will not return until quite message is received from the Client).

  iii. Call the *cleanup()* method to cleanup and then exit the program.



```
/**
 * Entry method for the Server application (for testing only).
 *
 * @param args
 * @throws IOException
 */
public static void main(String[] args) throws IOException
{
    // Create an instance of this Server
    //  Start the Server on port 6666 (which will not return until the Shutdown Command is received)
    //  and then on exit clean everything up
    Server server = new Server();
    server.start(6666);
    server.cleanup();
}
```

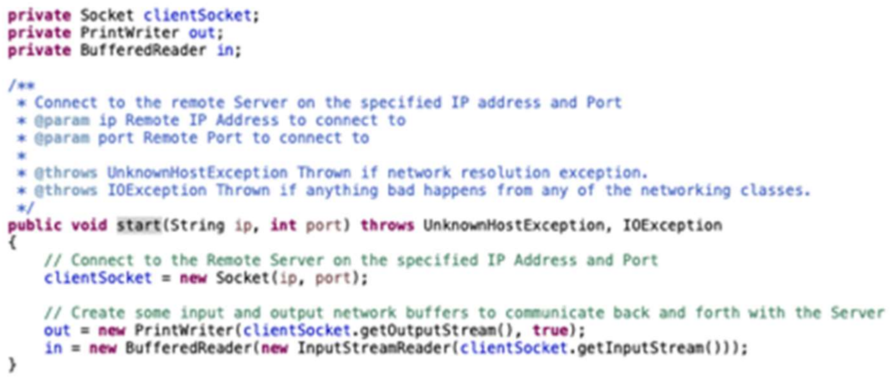
2. Create a new class *Client* in the *app* package with a *main()* method to send and process response messages.

 a. Create a method named *start()* method that takes an IP address and port as parameters, returns void, and throws IOException.

  i. Create a *Socket* to connect to the specified IP address and port.

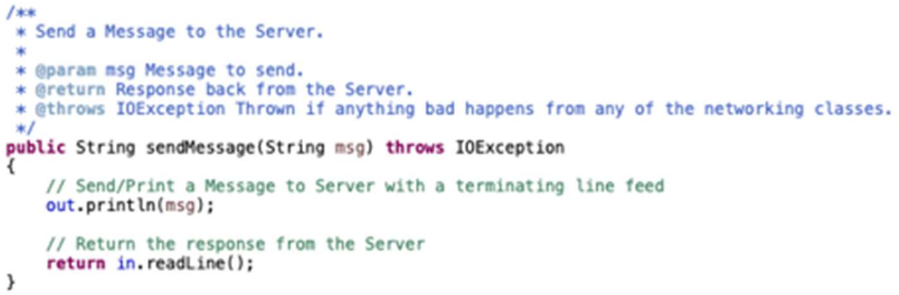
  ii. Create a *PrintWriter* for sending text over the socket to the Server.

  iii. Create a *BufferReader* for receiving text over the socket from the Server.



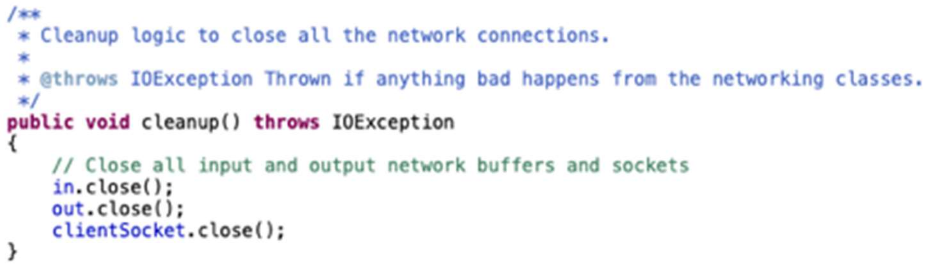
```
private Socket clientSocket;
private PrintWriter out;
private BufferedReader in;

/**
 * Connect to the remote Server on the specified IP address and Port
 * @param ip Remote IP Address to connect to
 * @param port Remote Port to connect to
 *
 * @throws UnknownHostException Thrown if network resolution exception.
 * @throws IOException Thrown if anything bad happens from any of the networking classes.
 */
public void start(String ip, int port) throws UnknownHostException, IOException
{
    // Connect to the Remote Server on the specified IP Address and Port
    clientSocket = new Socket(ip, port);

    // Create some input and output network buffers to communicate back and forth with the Server
    out = new PrintWriter(clientSocket.getOutputStream(), true);
    in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
}
```

b. Create a method named *sendMessage()* method that a string as a message parameter, returns the Server response, and throws IOException.
   i. Use the created *PrintWriter* to send the specified message to the Server.
   ii. Return the Server response by reading from the created BufferReader.



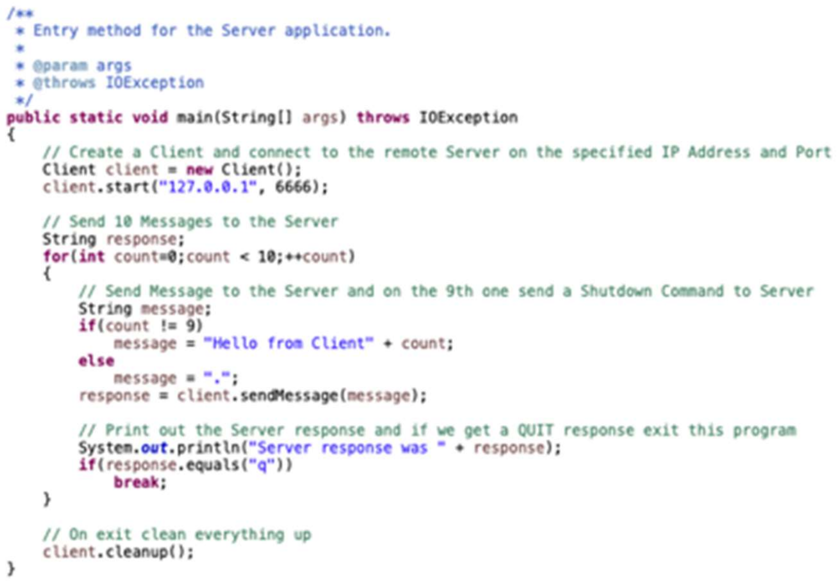
```
/**
 * Send a Message to the Server.
 *
 * @param msg Message to send.
 * @return Response back from the Server.
 * @throws IOException Thrown if anything bad happens from any of the networking classes.
 */
public String sendMessage(String msg) throws IOException
{
    // Send/Print a Message to Server with a terminating line feed
    out.println(msg);

    // Return the response from the Server
    return in.readLine();
}
```

c. Create a method named *cleanup()* method that takes no parameters, returns void, and throws IOException.
   i. Close all network buffers and sockets.



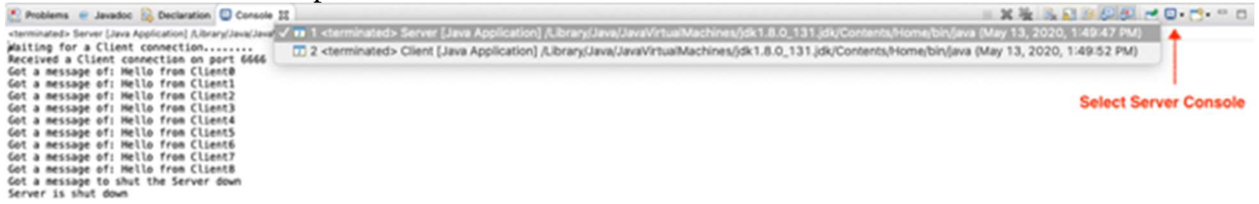
```
/**
 * Cleanup logic to close all the network connections.
 *
 * @throws IOException Thrown if anything bad happens from the networking classes.
 */
public void cleanup() throws IOException
{
    // Close all input and output network buffers and sockets
    in.close();
    out.close();
    clientSocket.close();
}
```

d. Implement the *main()* method.
   i. Create an instance of the *Client* class to use the local computer IP address and port 6666.
   ii. Call the *start()* method on the Client to connect to the Server.
   iii. Send 9 message to the Client and on the 9th method send the quit message ("."). Print the response to the console for each message sent to the Server.
   iv. Call the *cleanup()* method to cleanup and then exit the program.



```
/**
 * Entry method for the Server application.
 *
 * @param args
 * @throws IOException
 */
public static void main(String[] args) throws IOException
{
    // Create a Client and connect to the remote Server on the specified IP Address and Port
    Client client = new Client();
    client.start("127.0.0.1", 6666);

    // Send 10 Messages to the Server
    String response;
    for(int count=0;count < 10;++count)
    {
        // Send Message to the Server and on the 9th one send a Shutdown Command to Server
        String message;
        if(count != 9)
            message = "Hello from Client" + count;
        else
            message = ".";
        response = client.sendMessage(message);

        // Print out the Server response and if we get a QUIT response exit this program
        System.out.println("Server response was " + response);
        if(response.equals("q"))
            break;
    }

    // On exit clean everything up
    client.cleanup();
}
```

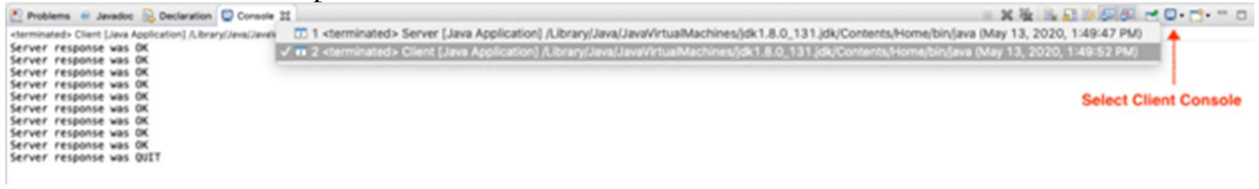
4. Run the *Server* application.
5. Run the *Client* application.
6. Observe the Server console output by selecting the Server console from the correct Console window in Eclipse. Take a screenshot.



```
<terminated> Server [Java Application] /Library/Java/Java
Waiting for a Client connection........
Received a Client connection on port 6666
Got a message of: Hello from Client0
Got a message of: Hello from Client1
Got a message of: Hello from Client2
Got a message of: Hello from Client3
Got a message of: Hello from Client4
Got a message of: Hello from Client5
Got a message of: Hello from Client6
Got a message of: Hello from Client7
Got a message of: Hello from Client8
Got a message to shut the Server down
Server is shut down
```

```
✓ 1 <terminated> Server [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java (May 13, 2020, 1:49:47 PM)
  2 <terminated> Client [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java (May 13, 2020, 1:49:52 PM)
```

7. Observe the Server console output by selecting the Server console from the correct Console window in Eclipse. Take a screenshot.



```
<terminated> Client [Java Application] /Library/Java/Java
Server response was OK
Server response was OK
Server response was OK
Server response was OK
Server response was OK
Server response was OK
Server response was OK
Server response was OK
Server response was OK
Server response was QUIT
```

```
  1 <terminated> Server [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java (May 13, 2020, 1:49:47 PM)
✓ 2 <terminated> Client [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java (May 13, 2020, 1:49:52 PM)
```

8. Provide a brief (3-4 sentences) description for each of the networking classes that were used and how they were used to create a network client server application.

Deliverables:

The following needs to be submitted as this part of the Activity:

a. Theory of operation write ups.
b. All screenshots of application in operation.
c. ZIP file of the code in the project folder. Include the JavaDoc generated for the project.

Part 3: Multi-threaded Client and Server Networking Application

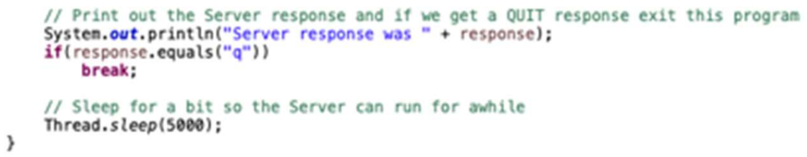
Overview

Goal and Directions:

In this activity will make the synchronous server application created in the previous activity asynchronous by using Java Threads.

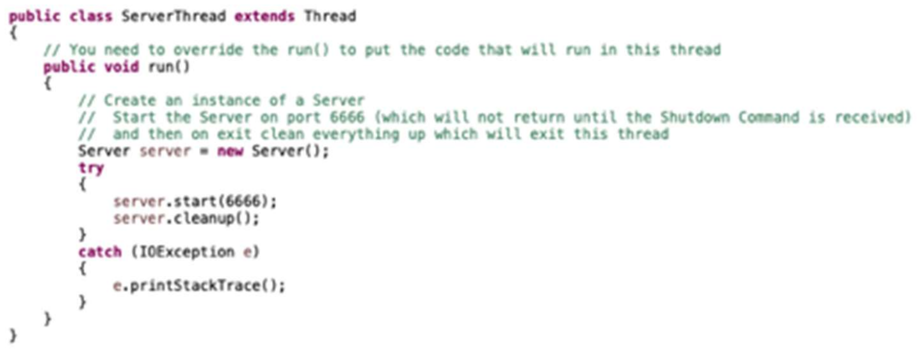
Execution

1. Create a new project named *topic6-3*. Copy all the code from *topic6-2* to the new project. Run the new *topic6-3* project to ensure the new project is working properly.
2. Update the *main()* method in the *Client* class to sleep for 5 seconds in between sending messages (this is only so you can see the program actually run).



```
        // Print out the Server response and if we get a QUIT response exit this program
        System.out.println("Server response was " + response);
        if(response.equals("q"))
            break;

        // Sleep for a bit so the Server can run for awhile
        Thread.sleep(5000);
    }
```

3. Create a new class *ServerThread* in the *app* package that extends from the *Thread* class. In the *run()* method create an instance of a *Server*, start the server on port 6666, and then cleans up after the server returns.



```
public class ServerThread extends Thread
{
    // You need to override the run() to put the code that will run in this thread
    public void run()
    {
        // Create an instance of a Server
        //  Start the Server on port 6666 (which will not return until the Shutdown Command is received)
        //  and then on exit clean everything up which will exit this thread
        Server server = new Server();
        try
        {
            server.start(6666);
            server.cleanup();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

4. Create a new class *ServerApp* in the *app* package with a *main()* method. In the *main()* method create an instance of a *ServerThread,* starts the thread, and then sits in a loop while the thread is still running printing out a dot (".") to the console and sleeps for 5 seconds. This will enable you to see both the *ServerApp* and *Server* classes output on the console validating that truly indeed both applications are running (the Server is no longer synchronous to the client running the Server).
5. Run the *ServerApp* application.
6. Run the *Client* application.
7. Observe the *ServerApp* and *Client* consoles output by selecting the Server console from the correct Console window in Eclipse. Take a screenshot of both console windows.

8. Provide a brief (3-4 sentences) description using Java Threads improved the design and execution of the Server.

Deliverables:

The following needs to be submitted as this part of the Activity:

a. Theory of operation write ups.
b. All screenshots of application in operation.
c. ZIP file of the code in the project folder. Include the JavaDoc generated for the project.

Research Questions

1. Research Questions: For traditional ground students in a Word document answer the following questions:
    a. Explore the NIO classes in the Java programming language. Explain what features are provided by these classes and how they can be used to write network-based applications. Summarize your answers and rational in 300 words.
    b. A number of the networking classes are synchronous in nature. Explain how using multi-threaded programming techniques can make the code behave asynchronously in an application and improve performance. Summarize your answers and rational in 300 words.

Final Activity Submission

1. In a Microsoft Word document complete the following for the Activity Report:
    a. Cover Sheet with the name of this assignment, date, and your name.
    b. Section with a title that contains all the diagrams, screenshots, and theory of operation writeups.
    c. Zip file with all code and generated JavaDoc documentation files.
    d. Section with a title that contains the answers to the Research Questions (traditional ground students only).
2. Submit the Activity Report and zip file of the Code and Documentation to the Learning Management System (LMS).