CST-239 Activity 6 Guide

Contents

[Part 1: Creating a Thread 1](#_Toc40184236)

[Part 2: Creating a Client and Server Networking Application 1](#_Toc40184237)

[Part 3: Multi-threaded Client and Server Networking Application 1](#_Toc40184238)

# 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 as 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:
   1. Extend from java.lang.Thread class.
   2. Implement the void run() method by simply printing a message to the console of the name of the *MyThread1* class.

A screenshot of a cell phone

Description automatically generated

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



1. Create a new class named *TestMyThreads* class in the *app* package with a *main*(). Implement the logic in the *main()* as follows:
   1. Create an instance of the *MyThread1* class and call the *start()* method on the class.
   2. 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.

A screenshot of a cell phone

Description automatically generated

1. Run the *TestMyThreads* application.
2. Take a screenshot of the console output.
3. Remember, the threads are sharing the CPU time when running the application code, the code in MyThread1, and MyThread2. Provide a brief (3- to 4-sentence) description of how and why the output was displayed.

Part 2 – More Complex Threads

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:
   1. Add a for loop in the run() method for up to 100 iterations and that for each iteration:
      1. 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).

A picture containing knife

Description automatically generated

1. Modify the *MyThread2* class as follows.
   1. Add a for loop in the run() method for up to 100 iterations and that for each iteration:
      1. 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.
2. Run the *TestMyThreads* application.
3. Take a screenshot of the console output.
4. 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- to 4-sentence) description of how and why the output was displayed.
5. 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.



1. Run the *TestMyThreads* application. Click the Stop icon in the console to stop the execute of the application.
2. Take a screenshot of the console output.
3. 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- to 4-sentence) description of how and why the output was 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.

A screenshot of a cell phone

Description automatically generated

1. 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*.

A screenshot of a cell phone

Description automatically generated

1. 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.

A screenshot of a cell phone

Description automatically generated

1. Run the *CounterWorker* application.
2. Take a screenshot of the console output.
3. 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- to 4-sentence) description of how and why the output was displayed.
4. Let’s fix the concurrency problem in the multi-threaded code.
5. Make the access to incrementing the counter thread safe by making the method synchronized using the Java synchronized keyword.



1. Run the *CounterWorker* application.
2. Take a screenshot of the console output.
3. The end result, if execution is now correct, should end 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- to 4-sentence) description of how and why the output was displayed.

Deliverables:

The following need to be submitted as this part of the activity:

1. Theory of operation write-ups.
2. All screenshots of application in operation.
3. ZIP file of the code in the project folder. Include the JavaDoc generated for the project.

# 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 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:
   1. Create a method named *start()* method that takes a port as parameter, returns void, and throws IOException.
      1. Create a *ServerSocket* to connect to the specified port.
      2. Wait for a connection from the Client by calling *accept()* on the socket.
      3. Create a *PrintWriter* for sending text over the socket to the Client.
      4. Create a *BufferReader* for receiving text over the socket from the Client.
      5. 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 a quit message is received, send back a quit (“QUIT”) message to the Client and break out the loop to return to the *main()* method.

A screenshot of a social media post

Description automatically generated

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

A screenshot of a cell phone

Description automatically generated

* 1. Implement the *main()* method.
     1. Create an instance of the *Server* class.
     2. Call the *start()* method on the Server (this will not return until quite message is received from the Client).
     3. Call the *cleanup()* method to cleanup and then exit the program.

A screenshot of a cell phone

Description automatically generated

1. Create a new class *Client* in the *app* package with a *main()* method to send and process response messages.
   1. Create a method named *start()* method that takes an IP address and port as parameters, returns void, and throws IOException.
      1. Create a *Socket* to connect to the specified IP address and port.
      2. Create a *PrintWriter* for sending text over the socket to the Server.
      3. Create a *BufferReader* for receiving text over the socket from the Server.

A screenshot of a social media post

Description automatically generated

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

A screenshot of a cell phone

Description automatically generated

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

A screenshot of a cell phone

Description automatically generated

* 1. Implement the *main()* method.
     1. Create an instance of the *Client* class to use the local computer IP address and port 6666.
     2. Call the *start()* method on the Client to connect to the Server.
     3. 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.
     4. Call the *cleanup()* method to cleanup and then exit the program.



1. Run the *Server* application.
2. Run the *Client* application.
3. Observe the Server console output by selecting the Server console from the correct Console window in Eclipse. Take a screenshot.

A screenshot of a social media post

Description automatically generated

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

A screenshot of a cell phone

Description automatically generated

1. Provide a brief (3- to 4-sentence) 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 need to be submitted as this part of the activity:

1. Theory of operation write-ups.
2. All screenshots of application in operation.
3. 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, you 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).

A screenshot of a cell phone

Description automatically generated

1. 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 clean up after the server returns.

A screenshot of a cell phone

Description automatically generated

1. 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 indeed both applications are running (the Server is no longer synchronous to the client running the Server).
2. Run the *ServerApp* application.
3. Run the *Client* application.
4. 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.
5. Provide a brief (3- to 4-sentence) description of how using Java Threads improved the design and execution of the Server.

Deliverables:

The following need to be submitted as this part of the activity:

1. Theory of operation write-ups.
2. All screenshots of application in operation.
3. ZIP file of the code in the project folder. Include the JavaDoc generated for the project.

**Research Questions**

1. Research Questions: Online students will address these in the Discussion Forum and traditional on ground students will address them in this assignment.
   1. 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 rationale in 300 words.
   2. 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 rationale in 300 words.

**Final Activity Submission**

1. In a Microsoft Word document, complete the following for the Activity Report:
   1. Cover sheet with the name of this assignment, date, and your name.
   2. Section with a title that contains all the diagrams, screenshots, and theory of operation write-ups.
   3. Zip file with all code and generated JavaDoc documentation files.
   4. 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).