**CST-239 Activity 6**

**Sept 24,2023**

**Taofik Otiotio**

**PART 1 OF 1: CREATING A THREAD**

**THEORY OF OPERATION**

Multithreading in Java involves the concurrent execution of multiple sequential operations, or threads, to improve resource utilization and responsiveness. The Thread class can be extended to create new threads, with the run() method overridden to define the thread's behavior. For example, the MyThread1 class extends the Thread class and overrides the run method to print a message, iteration numbers, and introduce a 1 second sleep between iterations.

The Runnable interface allows any class to make its instances capable of executing in a separate thread. The MyThread2 class implements the Runnable interface and defines the run method.

In the TestMyThreads class, instances of MyThread1 and MyThread2 are created, with MyThread1 being run using its start() method. MyThread2 is started using the start() method for its instance. Both MyThread1 and MyThread2 use the Thread.sleep() method to introduce a pause in their execution, often used to simulate real-world scenarios where tasks might take time or introduce delays.

**A SCREEN SHOT OF THE CONSOLE OUTPUT**

A computer screen with a message

Description automatically generated

**PROVIDE A BRIEF (3-4 SENTENCES) DESCRIPTION HOW AND WHY THE OUTPUT GOT DISPLAYED**

The TestMyThreads class executes its main method, initiating two threads: thread1 (MyThread1) and thread2 (Thread targeting MyThread2). These threads execute their run methods, generating messages to the console. However, the order of these messages is not deterministic, as threads run concurrently, sharing CPU time based on the thread scheduler's decisions. This can result in messages from MyThread1 appearing first or from MyThread2 appearing first.

**PART 1 OF 2 – MORE COMPLEX THREADS**

**A SCREEN SHOT OF THE CONSOLE OUTPUT**

A screenshot of a computer

Description automatically generated

**PROVIDE A BRIEF (3-4 SENTENCES) DESCRIPTION HOW AND WHY THE OUTPUT GOT DISPLAYED**

Multi-threaded programs have no guarantee on the order of thread execution due to the thread scheduler in the operating system. Threads may not always run in the order they are started and can interleave. Factors like thread priority, operating system scheduling, and output stream caching can cause delays in displaying output. Even if MyThread1 completes all iterations before MyThread2, running the program multiple times may yield different results due to the concurrent nature of threads, with iterations interleaving in a non-deterministic order.

**A SCREEN SHOT OF THE CONSOLE OUTPUT**

A screenshot of a computer

Description automatically generated

**PROVIDE A BRIEF (3-4 SENTENCES) DESCRIPTION HOW AND WHY THE OUTPUT GOT DISPLAYED**

In the TestMyThreads main method, MyThread1 and MyThread2 execute their run methods concurrently, competing for CPU time. MyThread1 sleeps for 1000 milliseconds between iterations, while MyThread2 sleeps for 500 milliseconds. This difference in sleep durations leads to MyThread2 printing its messages at twice the frequency of MyThread1, producing two messages for every message MyThread1 outputs. The output displays a mix of "MyThread1 is running iteration..." and "MyThread2 is running iteration...", with MyThread2 messages appearing twice as often in quick succession before MyThread1's, indicating faster performance. The sequence may vary based on OS thread scheduler, system load, and other factors, but the pattern should be noticeable due to sleep durations.

**PART 1OF 3 – THREADS WITH A CONCURRENCY PROBLEM AND FIXING THE PROBLEM**

**A SCREEN SHOT OF THE CONSOLE OUTPUT**

A screenshot of a computer

Description automatically generated

**PROVIDE A BRIEF (3-4 SENTENCES) DESCRIPTION HOW AND WHY THE OUTPUT GOT DISPLAYED**

The discrepancy in the final counter value, even though 1000 threads are incrementing the counter, arises due to a race condition in the increment method of the Counter class. This method is not thread-safe, which means that multiple threads can access and modify the count simultaneously. Imagine two threads reading the count value as 5 at the same time. Both threads increment it independently and update the value to 6, even though they should've collectively incremented it to 7. To prevent such issues, one needs to employ synchronization techniques, such as the “**synchronized”**.

**A SCREEN SHOT OF THE CONSOLE OUTPUT**

A screenshot of a computer

Description automatically generated

**PROVIDE A BRIEF (3-4 SENTENCES) DESCRIPTION HOW AND WHY THE OUTPUT GOT DISPLAYED**

The previous version's conditions issue was resolved by synchronizing the Counter class's increment method. This ensures that only one thread can access and execute this method at a time, serializing access to the shared resource, the count variable. This prevents other threads from blocking the current thread until it completes, ensuring that all 1000 threads safely increment the counter one by one, resulting in the expected final counter value of 1000.

**PART 2: CREATING A CLIENT AND SERVER NETWORKING APPLICATION**

**THEORY OF OPERATION**

The application is bifurcated into two primary components: the Server and the Client. The Server establishes a listening socket, waiting for clients to connect. Once a connection is initiated, communication channels are established, facilitating the exchange of messages between the two. The Client, conversely, actively seeks to establish a connection to the server, send messages, and process server responses.

**Operational Details:**

1. **Server Operations**:
   * **Initialization**: The server's essential attributes, primarily networking components like the ServerSocket, are initialized upon instantiation.
   * **Connection Handling**: Upon invoking the start(int port) method, the server listens on the designated port. When a client seeks a connection, the server acknowledges it using the accept() method, thereby setting up the communication channels.
   * **Message Exchange**: Messages from the client are read using a BufferedReader. If a message equals ".", the server responds with "QUIT" and preps for shutdown. All other messages receive an "OK" response. This ensures the server remains responsive throughout the session.
   * **Resource Management**: Post communication, the cleanup() method is employed to ensure all network connections and buffers are closed to free up resources.
2. **Client Operations**:
   * **Initialization**: The client initializes essential attributes and prepares for communication upon instantiation.
   * **Server Communication**: Using the start(String ip, int port) method, the client tries to connect to the server specified by the IP and port. Successful connections result in the initialization of communication channels.
   * **Message Transactions**: The client can send messages to the server via the sendMessage(String msg) method. Each sent message anticipates a corresponding response from the server, which is then processed.
   * to **Resource Management**: After the communication session, the client invokes the cleanup() method close all network resources efficiently.
3. **Application Lifecycle**:
   * **Server's Lifecycle**: Initiated via its main method, the server starts by listening on port 6666 (for illustration). Upon the end of its session, cleanup procedures are executed to close network connections.
   * **Client's Lifecycle**: The client's main method sets the stage for its operation. The client establishes a connection to a server (in this example, at IP "127.0.0.1" on port 6666), sends a sequence of messages, processes server responses, and concludes with cleanup procedures.

**A SCREEN SHOT OF THE CONSOLE OUTPUT**

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

**PROVIDE A BRIEF (3-4 SENTENCES) DESCRIPTION HOW AND WHY THE OUTPUT GOT DISPLAYED**

When the application runs, the Server begins by listening for client connections on port 6666. Upon the Client initiating a connection, the Server acknowledges it and waits for messages. For each of the first 9 messages the Client sends (which are greetings followed by a number), the Server responds with "OK" and logs the received message. When the Client sends the tenth message as ".", the Server replies with "QUIT", signaling its intent to shut down, and then closes the connection.

**PART 3: MULTI-THREADED CLIENT AND SERVER NETWORKING APPLICATION**

**THEORY OF OPERATION**

The server-client communication system is a system that consists of a client application that initializes its socket-related resources, sets up an IP address and port number, and initiates a connection. The server then listens for incoming client connections and notifies the operator of the connection. When a client tries to connect, the server establishes a socket connection, enabling bi-directional communication.

The client sends a series of messages to the server in a loop, sending "Hello from Client [iteration number]" for nine iterations, and then sending a "." message signaling the server to shut down. The server echoes an "OK" response for each received message, but responds with "QUIT" when it receives the "." message. The server also prints every message it receives, providing a log of client communication.

A feedback mechanism is used by the server, which prints a dot every 5 seconds to notify the operator that the server is actively running and monitoring even if no clients are currently connected. After sending all its messages, the client initiates a cleanup procedure, closing its communication channels and sockets. The server acknowledges the request to terminate by sending back a "QUIT" message and closes all open resources, ensuring a graceful shutdown.

**A SCREEN SHOT OF THE CONSOLE OUTPUT**

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

**PROVIDE A BRIEF (3-4 SENTENCES) DESCRIPTION HOW AND WHY THE OUTPUT GOT DISPLAYED**

The server monitors client connections and prints notifications about their status and received messages. It waits for a client connection and prints "Waiting for a Client connection..." until a client connects. If a client sends a message, it prints "Got a massage of: [message]". If a client sends a ".", it signals termination. The ServerApp prints a dot to the console every 5 seconds, indicating the server is operational and monitoring connections or messages.