Activity 6

Jesse Dalton
March 5, 2023

PART 1a

Threads Output:

```
MyThread1 is running
MyThread2 is running
```

I learned from this activity the difference between calling run() and start() on a Thread class. Run executes the specified method synchronously. Start creates a new thread, then executes the run method in that thread asynchronously. In this example, Thread1 is started first, so it outputs first, however, technically both classes are being executed concurrently in their own threads.

PART 1b

Threads Output:

```
MyThread1 is running 0
MyThread1 is running 1
MyThread1 is running 2
MyThread1 is running 3
MyThread1 is running 4
MyThread1 is running 5
MyThread1 is running 6
MyThread1 is running 7
MyThread2 is running 0
MyThread1 is running 8
MyThread2 is running 1
MyThread1 is running 9
MyThread2 is running 2
MyThread1 is running 10
MyThread1 is running 11
MyThread1 is running 12
MyThread2 is running 3
MyThread1 is running 13
MyThread2 is running 4
MyThread1 is running 14
MyThread2 is running 5
MyThread1 is running 15
MyThread2 is running 6
MyThread1 is running 16
MyThread1 is running 17
MyThread2 is running 7
MyThread1 is running 18
MyThread2 is running 8
MyThread1 is running 19
MyThread2 is running 9
```

The spec says the output here should show all of Thread1 running first, then Thread2, but mine does not. More of Thread1 is run before Thread2 starts, and then Thread2 finishes after Thread1, but they are mostly concurrent. This is because both are being run simultaneously on separate threads.

Threads Output with sleep code:

```
MyThread1 is running 0
MyThread2 is running 0
MyThread2 is running 1
MyThread2 is running 2
MyThread1 is running 1
MyThread2 is running 3
MyThread2 is running 4
MyThread1 is running 2
MyThread2 is running 5
MyThread1 is running 3
MyThread2 is running 6
MyThread2 is running 7
MyThread1 is running 4
MyThread2 is running 8
MyThread1 is running 5
MyThread2 is running 9
MyThread2 is running 10
MyThread1 is running 6
MyThread2 is running 11
MyThread2 is running 12
MyThread1 is running 7
MyThread2 is running 13
MyThread1 is running 8
MyThread2 is running 14
MyThread2 is running 15
MyThread1 is running 9
MyThread2 is running 16
MyThread2 is running 17
MyThread1 is running 10
MyThread2 is running 18
MyThread1 is running 11
MyThread2 is running 19
MyThread2 is running 20
MyThread2 is running 21
MyThread1 is running 12
MyThread2 is running 22
MyThread1 is running 13
MyThread2 is running 23
MyThread2 is running 24
```

With the added sleep code, Thread1 takes longer to run than Thread2. Each iteration of Thread1 is twice as long as Thread2, and we see that the distribution of Thread2 output statements is heavier in the beginning, while near the end of execution, it is only Thread1 statements because Thread2 finished earlier.

PART 1c

Threads Output:

```
Initial count: 0
Final count: 992
```

Threads Output different trial:

```
Initial count: 0
Final count: 998
```

This program is an example of a race condition. Race conditions are situations where the procedural outcome of a program is dependent on the situational state of the program during runtime. Race conditions often cause the program to behave in an unexpected way. In this example, although we expect the counter to reach 1000, it is almost always just shy. This can be explained by the race condition causing different threads to read and write to the counter at non-sequential times.

In order to increment, the program must first read the counter. Once read by a given thread, the counter is incremented by that thread, then the new value is written back to the counter. If one thread accesses the counter value, then another thread accesses the counter value *before* the first thread has had a chance to update and write the new value, then the second thread will receive the same value as the first. This means that both threads start with the same value and end up writing the same value to the counter. Effectively an increment gets skipped.

Threads Output synchronized:

```
Initial count: 0
Final count: 1000
```

The synchronized keyword can be applied to methods and blocks of code. It makes the code threadsafe by providing a monitor object. The JVM is configured to not allow more than one thread to access a monitor object at any given time.

PART 2

Server Output:

```
Server listening on port 8080...
Received a Client connection on port 8080
Client said: Hello from client 0
Client said: Hello from client 1
Client said: Hello from client 2
Client said: Hello from client 3
Client said: Hello from client 4
Client said: Hello from client 5
Client said: Hello from client 6
Client said: Hello from client 7
Client said: Hello from client 8
Client said: Hello from client 9
Client said: .
Shutting down
```

Client Output

```
Server said: OK
Server said: OK
Server said: OK
Server said: OK
Server said: OK
Server said: OK
Server said: OK
Server said: OK
Server said: OK
Server said: OK
Server said: QUIT
```

The Server class uses a ServerSocket, which establishes a server socket on the given port on that machine. This socket listens for incoming traffic. Once a client socket is accepted, the input and output streams on the socket object can be used to communicate with the client socket.

The Client class uses a simple Socket object, which also establishes a socket connection using the specified port and hostname. Also similarly, this socket can communicate using its input and output streams.

PART 3

ServerApp Output:

```
Server listening on port 8080...
Received a Client connection on port 8080
Client said: Hello from client 0
Client said: Hello from client 1
Client said: Hello from client 2
Client said: Hello from client 3
Client said: Hello from client 4
Client said: Hello from client 5
Client said: Hello from client 6
Client said: Hello from client 7
Client said: Hello from client 8
Client said: Hello from client 9
Client said: .
Shutting down
```

Client Output

```
Server said: OK
Server said: OK
Server said: OK
Server said: OK
Server said: OK
Server said: OK
Server said: OK
Server said: OK
Server said: OK
Server said: OK
Server said: QUIT
```

Adding threads to the program makes the program capable of handling multiple requests at the same time. Instead of blocking and waiting for one request to finish before starting the next one, the Server receives a thread and immediately creates a new thread to handle it. This also fixes the problem of requests being dropped while the server is busy.