Assignment 1
Topic: Java Thread & Sockets Basics

Exercise 1.1 – Distribution with Threads

Write a Java application that computes and outputs to the console which integer in the range 1 – 100000 has the largest number of divisors, and the number of divisors it has. The computation method does not have to be particularly efficient, you may use a simple brute force approach.

Run the application and measure the execution time. (e.g., by using *System.nanoTime()*).

Modify your application such that it uses multiple threads to solve this problem. In a first step, you should naively divide the problem into even parts and create one thread for each part. (For example, if you use 10 threads, thread 1 computes 1-10000, thread 2 computes 10001-20000, etc.). When all threads are finished, the main thread should combine the partial results (which can be stored in the Thread objects) and output the final result.

Run the application again (with different numbers of threads) and compare the execution times.

Exercise 1.2 – Load balancing

In the previous exercise, we divided up a large task into smaller subtasks and created a thread to execute each subtask. However, this might not be the most efficient way to do this. Because of the nature of the problem some threads have much more work to do than others - it is much easier to find the number of divisors of a small number than it is of a big number. A much better approach is to break up the problem into a fairly large number of smaller problems.

For that purpose, implement a thread pool to execute the tasks: Each thread in the pool runs in a loop in which it repeatedly takes a task (one number from the input range) from a thread-safe queue (e.g., a *ConcurrentLinkedQueue*) and computes the number of divisors for this number, until the task queue is empty.

Run the application again and compare the execution time with the previous version.

Exercise 1.3 – Producer/Consumer

In this exercise, we will further extend the application from the previous exercise. Besides the task queue, we will use a second queue for the results that are "produced" by the computation threads. This means that when a thread has finished the computation for a number, it should write the result into this result queue.
The main thread (which started the other threads) "consumes" the individual results from the result queue as soon as they become available and combines them to an overall result. The result queue has to be a blocking queue (e.g., a *LinkedBlockingQueue*), since the main thread will have to wait for results to become available. Note that the main thread knows the exact number of results that it expects to read from the queue, so it can do so in a for loop; when the for loop completes, the main thread knows that all the tasks have been executed.

Exercise 1.4 – Sockets

Extend the server code of the tutorial example to support the following mathematical operations:

- Calculation of all prime numbers to a given bound N;
- Calculation of a perimeter of a circle;
- Square root of a given integer number.

The client code should also be extended to parse and visualize the results from the server.

The format for inquiring the server should be:
"IP address and port number", "operation", "integer N/radius R"

Exercise 1.5 – Theory

Prepare a short presentation (3-5 slides) about the following topics:

- Discuss and contrast the general advantages and disadvantages of distributed versus centralized systems.
- Summarize the fundamental challenges that arise when building an application with a distributed architecture.