**Exercise 1**

1. Servers create threads to respond every client request because it’s more efficient. It can still accept client calls while another request is being executed. If you give every connection a thread, then a thread can stay idle between requests and that can cause excessive resource usage.
2. Condition objects encapsulate the waits and signals used in guarded methods. So, one advantage is that they can be used to simplify class design by off-loading waiting and notification mechanics. Disadvantages of condition objects are by careless implementations you can increase complexity of the system or it can lead to nested monitor problem.
3. SimpleConditionObject can still able to work without instance variables because methods are synchronized which means in case of synchronization it’s going to use the lock on the object’s monitor itself instead of a variable.
4. Permits and latches are special form of conditions. A permit counts the number of threads that may enter a critical section. A latch has an opened or a closed feature additional to normal condition. It’s natural to use them when a condition requires to use some additional properties that are not captured by Java’s object locks.

**Exercise 2**

1. For this specific samples, actually it doesn’t matter because there’s not a huge time gap between them. However, for efficieny I’d prefer using Future solution, because if you want to calculate more numbers or if you want to calculate a number which is taking so much time, then with Futures you will need to wait at most as long as the longest calculation time. However, with Early reply, you have to wait for all calculations one by one because they synchronize on host, because of that Futures can finish earlier than Early Replies.
2. Executors allow us to manage Threads. You can create thread pool executor which specifies number of at most runnning threads, or you can create serial executor which enables us to schedule threads.
3. Solution in FutureTaskExecDemo.java file.

**Exercise 3**

Only modified TheNest.java

**Exercise 4**

1. 4 physical cores / 8 threads – Intel Core i7 2630QM
2. Concurrent threads = 2, rounds = 10.000, average of 3 executions: 720 ms

Concurrent threads = 4, rounds = 10.000, average of 3 executions: 702 ms

Concurrent threads = 8, rounds = 10.000, average of 3 executions: 735 ms

Concurrent threads = 2, rounds = 100.000, average of 3 executions: 6894 ms

Concurrent threads = 4, rounds = 100.000, average of 3 executions: 6673 ms

Concurrent threads = 8, rounds = 100.000, average of 3 executions: 6683 ms

As a result, this implementation does not scale well, because increasing number of concurrent threads have nearly no affect of reducing execution time.

1. I found out that this execution time is the result of starting that much threads. It’s because calculation inside the CalculationThread is not computationally hard. So, increasing or reducing the number of threads don’t have any affect on results, because calculation is done so quickly. Also, there’s no need for a synchronization inside the calculation thread because BigDecimal is immutable and Vector collection is thread safe. In this case, results above are the time required for my system to start 10k or 100k threads.
2. The reason of the cost is starting that much threads. So, we can turn this program into a sequential program by removing threads. To do this we will comment out t.start(); And add line t.run(); So, this will be no different than creating an object and invoking the method of the object for calculation. Here are results of this chance:

rounds = 10.000, average of 3 executions: 86 ms

rounds = 100.000, average of 3 executions: 366 ms

1. For drop-in replacement, Riemann zeta function can be an alternative. However, I found out that Bailey-Borwein-Plouffe is very efficient.
2. Runtimes with identical parameters vary because using threads are expensive and thread management can change with runtime system.