```
package util;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.util.Iterator;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

/**
 * The Concurrent Class has static methods to run Tasks in parallel
 *
 * @author Gokulnath Haribabu
 *
 *
 */
public class Concurrent {

    /**
     * This generic function was created to simplify running of a Parallelizable
     * Task for a large dataset. It takes its inspiration from
     * 1.) pool.map in python and
     * 2.) pmap in erlang
     * However this method does not return any results back
     * It only processes the tasks in parallel
     *
     * This code snippet was derived after reading the following blog entry for
     * understanding thread pool
     * http://www.javacodegeeks.com/2011/12/using-threadpoolexecutor-to-parallelize.html
     *
     *
     * @param data
     *            - Is an iterator which feeds the constructor of the klass
     * @param klass
     *            - klass is a generic class which implements Runnable However
     *            when declaring a generic Class u can only specify it as
     *            extends ( as opposed to implements )
     *
     * For Example: Class Task implements Runnable Constructor<Data>
     * data = Iterator<Data>
     *
     * Typical Usage: Concurrent.runTasks(data, Task)
     *
     * @throws InvocationTargetException
     * @throws IllegalAccessException
     * @throws InstantiationException
     * @throws IllegalArgumentException
     * @throws Exception
     */
    public static void runTasks(Iterator<?> data,
                                Class<? extends Runnable> klass)
                        throws IllegalArgumentException, InstantiationException,
                               IllegalAccessException, InvocationTargetException,
                               Exception {
```

```
        int cpus = Runtime.getRuntime().availableProcessors();
        int scaleFactor = 1;
        int maxThreads = cpus * scaleFactor;
        maxThreads = (maxThreads > 0 ? maxThreads : 1);

        ExecutorService pool = new ThreadPoolExecutor(
                                maxThreads, // core thread pool size
                                maxThreads, // maximum thread pool size
                                1, // time in minutes to wait before resizing pool
                                TimeUnit.MINUTES,
                                // When all the threads are busy then block...
                                new ArrayBlockingQueue<Runnable>(maxThreads,true),
                                // ... On blocking make the idle main thread
                                // to process the Task Queue sequentially
                                new ThreadPoolExecutor.CallerRunsPolicy());

        long startTime = System.currentTimeMillis();

        // Assumption : The Klass has only one constructor
        // and the constructor has one argument
        Constructor<?> klassCon = klass.getConstructors()[0];

        while (data.hasNext()) {
            Runnable r = (Runnable) klassCon.newInstance(data.next());
            pool.submit(r);
        }
        pool.shutdown();
        try {
            if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
                // pool didn't terminate after the first try
                pool.shutdownNow();
            }
            if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
                // pool didn't terminate after the second try
            }
        } catch (InterruptedException ex) {
            pool.shutdownNow();
            Thread.currentThread().interrupt();
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Elapsed time - " + ((endTime - startTime) / 1000)
                + " secs");
    }

}
```