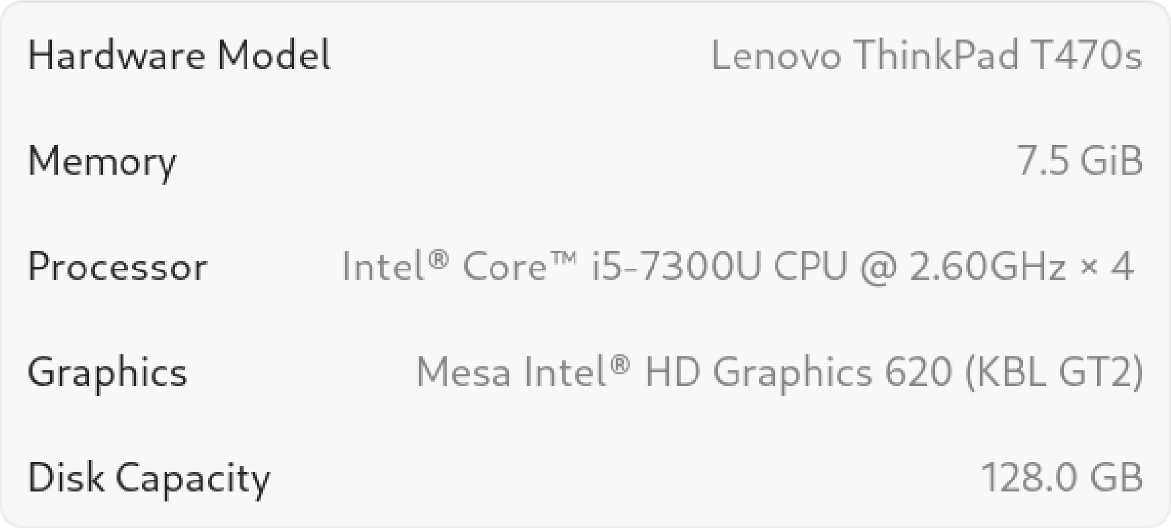
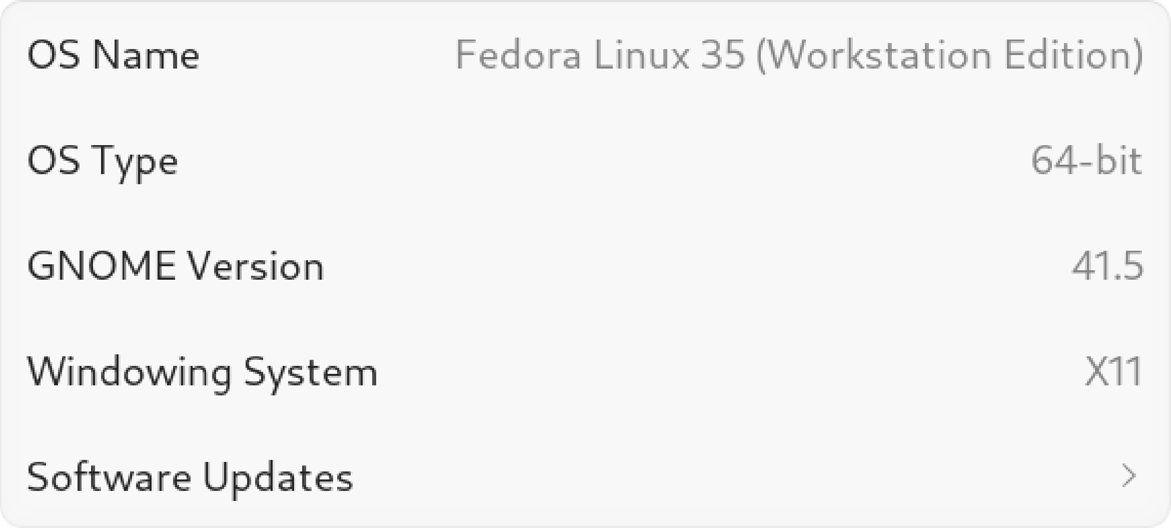
Multicore Computing Project 1

Hardware and Software Information

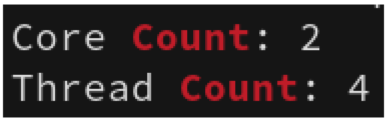


Hardware Model	Lenovo ThinkPad T470s
Memory	7.5 GiB
Processor	Intel® Core™ i5-7300U CPU @ 2.60GHz × 4
Graphics	Mesa Intel® HD Graphics 620 (KBL GT2)
Disk Capacity	128.0 GB



OS Name	Fedora Linux 35 (Workstation Edition)
OS Type	64-bit
GNOME Version	41.5
Windowing System	X11
Software Updates	›

Hyperthreading: ON



```
Core Count: 2
Thread Count: 4
```

Problem 1

Tables

Execution Times

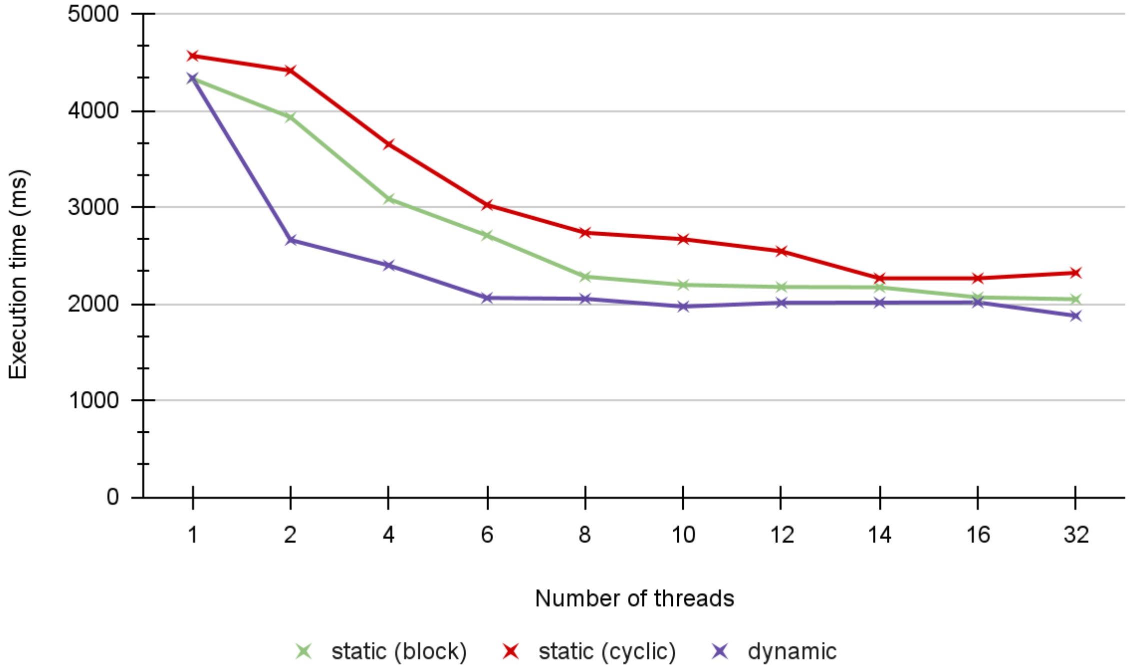
exec times in ms	1	2	4	6	8	10	12	14	16	32
static (block)	4330	3929	3085	2707	2280	2195	2173	2170	2066	2046
static (cyclic)	4566	4412	3650	3023	2735	2668	2543	2263	2263	2320
dynamic	4336	2660	2397	2060	2051	1971	2010	2012	2014	1875

Performance

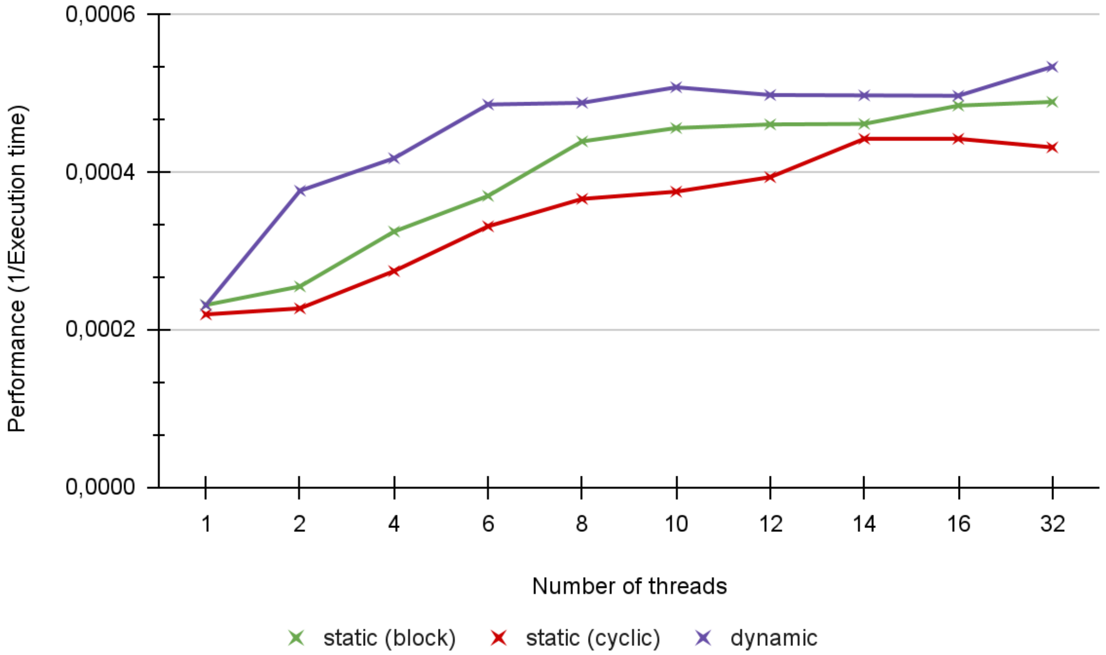
Performance 1/exec time	1	2	4	6	8	10	12	14	16	32
static (block)	0,0002309468822	0,000254517689	0,0003241491086	0,0003694126339	0,0004385964912	0,0004555808656	0,0004601932812	0,0004608294931	0,0004840271055	0,0004887585533
static (cyclic)	0,0002190100745	0,0002266545784	0,0002739726027	0,0003307972213	0,000365630713	0,0003748125937	0,000393236335	0,0004418912947	0,0004418912947	0,0004310344828
dynamic	0,0002306273063	0,0003759398496	0,0004171881519	0,0004854368932	0,0004875670405	0,0005073566717	0,0004975124378	0,0004970178926	0,0004965243297	0,0005333333333

Graphs

Execution Times



Performance



Interpretation

My interpretation of these results is that increasing the number of threads, significantly reduces execution times and therefore increases performance. However, it reaches a point of what I would call “peak necessary performance” where execution times and performance do not vary much when adding more threads. As we can see, execution times with 16 threads are rather low and when doubling the number threads results are pretty much the same.
As for comparing the three methods, I think dynamic load balancing is definitely the best approach to a problem like this one, when the number of threads is low. However when increasing the number of threads the three methods seem to achieve pretty close or identical, sometimes even better execution times. So I think the best approach would then come to which one uses less memory.

Compile & Run

Compile

To compile the code simply write:
javac [java_file]

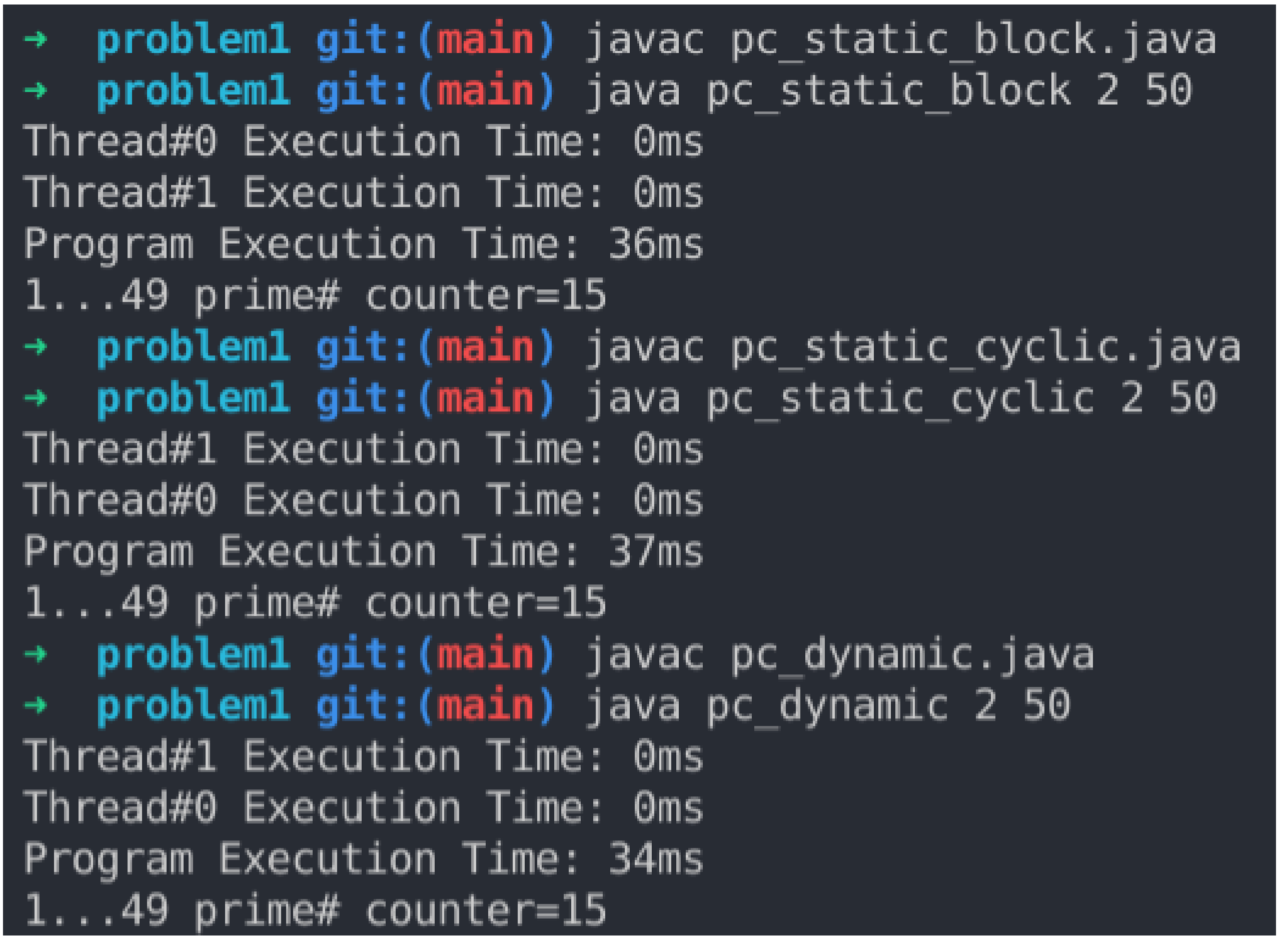
Replace “[java_file]” by the name of the desired java file

Run

To run simply write:
java [class_name] [nbThreads] [maxNb]

Replace:

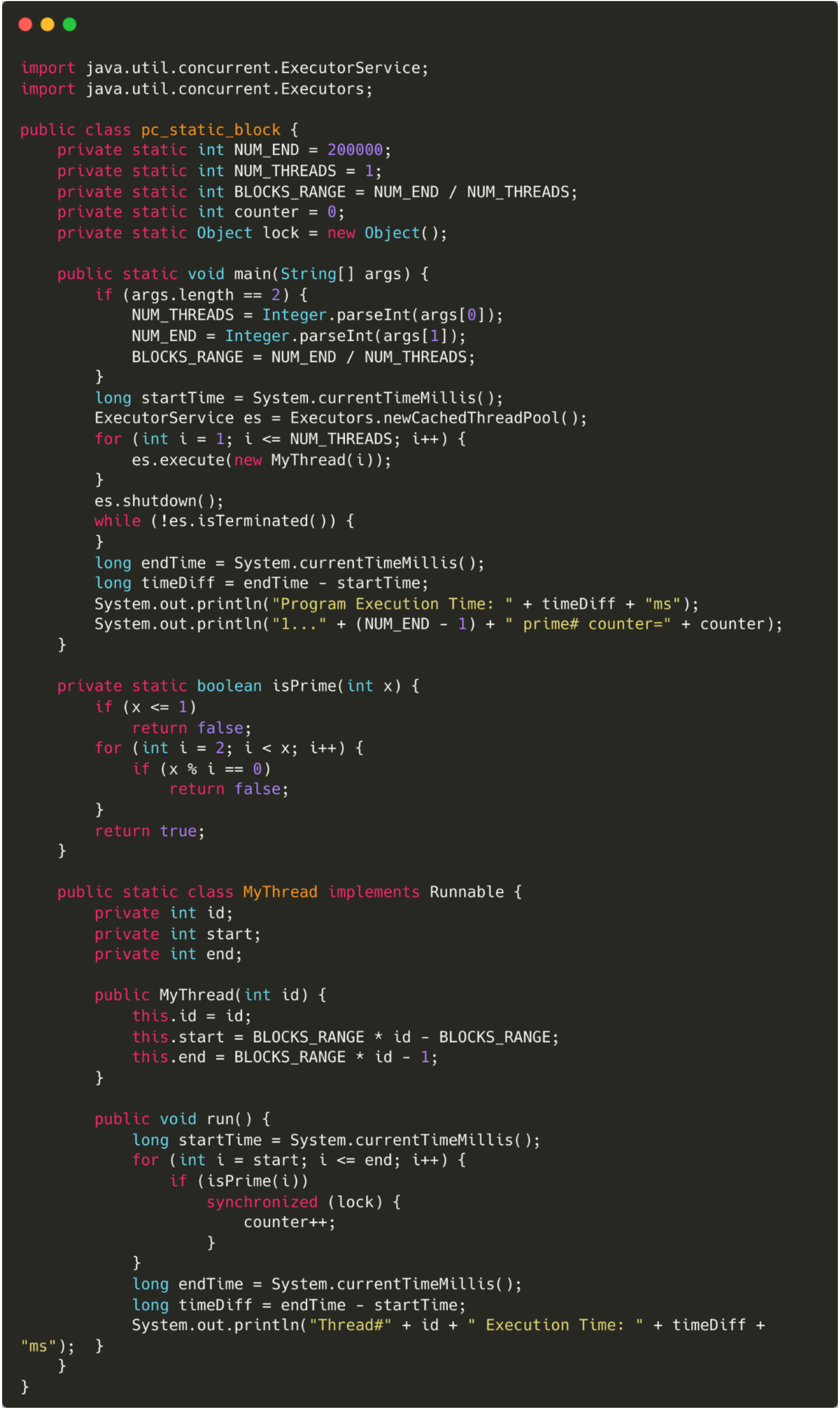
- [class_name] by the name of the compiled class
- [nbThreads] by the number of desired threads
- [maxNb] by the desired end number



```
➜  problem1 git:(main) javac pc_static_block.java
➜  problem1 git:(main) java pc_static_block 2 50
Thread#0 Execution Time: 0ms
Thread#1 Execution Time: 0ms
Program Execution Time: 36ms
1...49 prime# counter=15
➜  problem1 git:(main) javac pc_static_cyclic.java
➜  problem1 git:(main) java pc_static_cyclic 2 50
Thread#1 Execution Time: 0ms
Thread#0 Execution Time: 0ms
Program Execution Time: 37ms
1...49 prime# counter=15
➜  problem1 git:(main) javac pc_dynamic.java
➜  problem1 git:(main) java pc_dynamic 2 50
Thread#1 Execution Time: 0ms
Thread#0 Execution Time: 0ms
Program Execution Time: 34ms
1...49 prime# counter=15
```

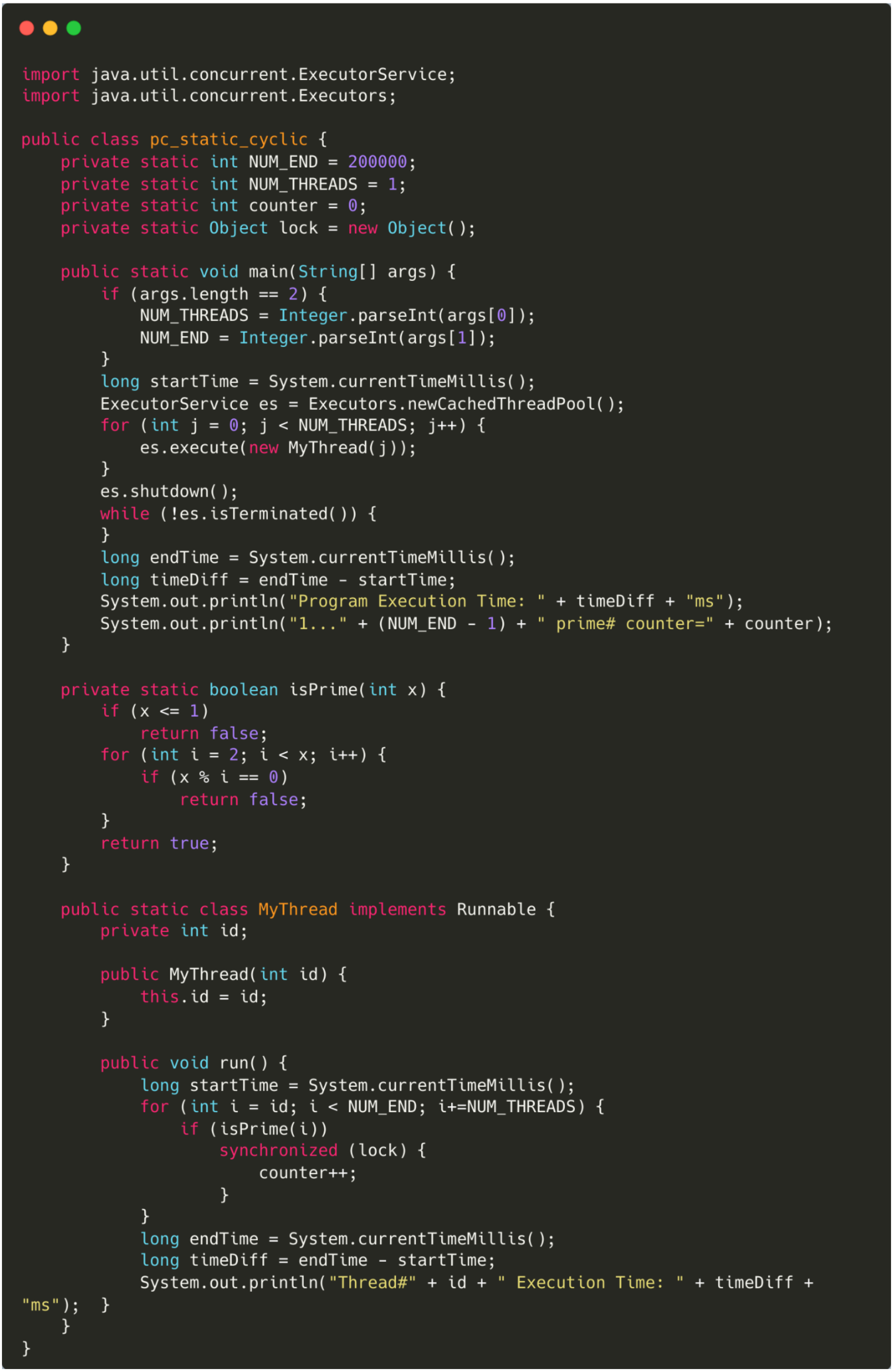
Code screenshots

Static block



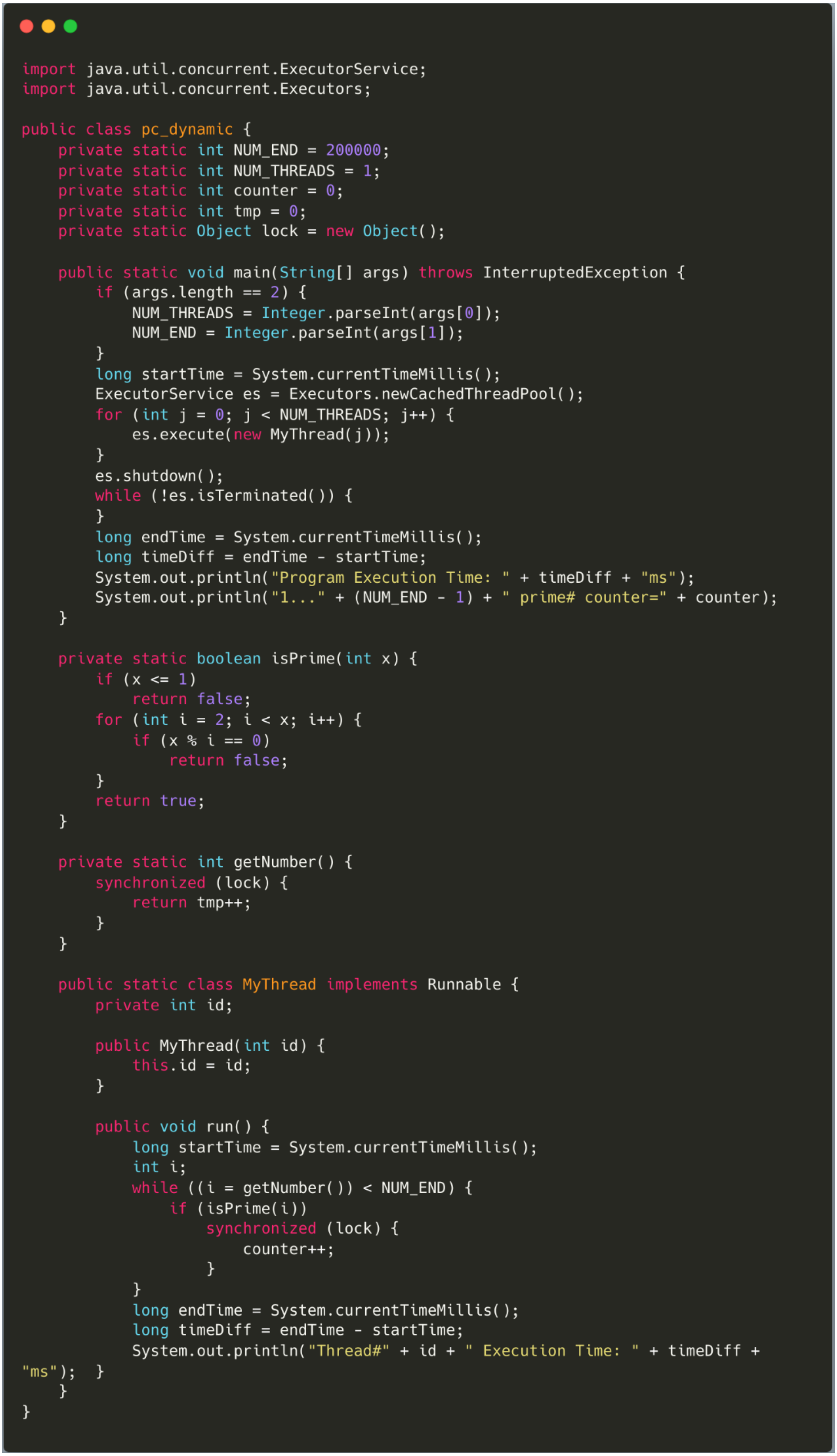
```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class pc_static_block {
    private static int NUM_END = 200000;
    private static int NUM_THREADS = 1;
    private static int BLOCKS_RANGE = NUM_END / NUM_THREADS;
    private static int counter = 0;
    private static Object lock = new Object();

    public static void main(String[] args) {
        if (args.length == 2) {
            NUM_THREADS = Integer.parseInt(args[0]);
            NUM_END = Integer.parseInt(args[1]);
            BLOCKS_RANGE = NUM_END / NUM_THREADS;
        }
        long startTime = System.currentTimeMillis();
        ExecutorService es = Executors.newCachedThreadPool();
        for (int i = 1; i <= NUM_THREADS; i++) {
            es.execute(new MyThread(i));
        }
        es.shutdown();
        while (!es.isTerminated()) {
        }
        long endTime = System.currentTimeMillis();
        long timeDiff = endTime - startTime;
        System.out.println("Program Execution Time: " + timeDiff + "ms");
        System.out.println("1..." + (NUM_END - 1) + " prime# counter=" + counter);
    }

    private static boolean isPrime(int x) {
        if (x <= 1)
            return false;
        for (int i = 2; i < x; i++) {
            if (x % i == 0)
                return false;
        }
        return true;
    }

    public static class MyThread implements Runnable {
        private int id;
        private int start;
        private int end;

        public MyThread(int id) {
            this.id = id;
            this.start = BLOCKS_RANGE * id - BLOCKS_RANGE;
            this.end = BLOCKS_RANGE * id - 1;
        }

        public void run() {
            long startTime = System.currentTimeMillis();
            for (int i = start; i <= end; i++) {
                if (isPrime(i))
                    synchronized (lock) {
                        counter++;
                    }
            }
            long endTime = System.currentTimeMillis();
            long timeDiff = endTime - startTime;
            System.out.println("Thread#" + id + " Execution Time: " + timeDiff +
"ms");  }
    }
}
```

Static cyclic



```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class pc_static_cyclic {
    private static int NUM_END = 200000;
    private static int NUM_THREADS = 1;
    private static int counter = 0;
    private static Object lock = new Object();

    public static void main(String[] args) {
        if (args.length == 2) {
            NUM_THREADS = Integer.parseInt(args[0]);
            NUM_END = Integer.parseInt(args[1]);
        }
        long startTime = System.currentTimeMillis();
        ExecutorService es = Executors.newCachedThreadPool();
        for (int j = 0; j < NUM_THREADS; j++) {
            es.execute(new MyThread(j));
        }
        es.shutdown();
        while (!es.isTerminated()) {
        }
        long endTime = System.currentTimeMillis();
        long timeDiff = endTime - startTime;
        System.out.println("Program Execution Time: " + timeDiff + "ms");
        System.out.println("1..." + (NUM_END - 1) + " prime# counter=" + counter);
    }

    private static boolean isPrime(int x) {
        if (x <= 1)
            return false;
        for (int i = 2; i < x; i++) {
            if (x % i == 0)
                return false;
        }
        return true;
    }

    public static class MyThread implements Runnable {
        private int id;

        public MyThread(int id) {
            this.id = id;
        }

        public void run() {
            long startTime = System.currentTimeMillis();
            for (int i = id; i < NUM_END; i+=NUM_THREADS) {
                if (isPrime(i))
                    synchronized (lock) {
                        counter++;
                    }
            }
            long endTime = System.currentTimeMillis();
            long timeDiff = endTime - startTime;
            System.out.println("Thread#" + id + " Execution Time: " + timeDiff +
"ms");  }
    }
}
```

Dynamic



```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class pc_dynamic {
    private static int NUM_END = 200000;
    private static int NUM_THREADS = 1;
    private static int counter = 0;
    private static int tmp = 0;
    private static Object lock = new Object();

    public static void main(String[] args) throws InterruptedException {
        if (args.length == 2) {
            NUM_THREADS = Integer.parseInt(args[0]);
            NUM_END = Integer.parseInt(args[1]);
        }
        long startTime = System.currentTimeMillis();
        ExecutorService es = Executors.newCachedThreadPool();
        for (int j = 0; j < NUM_THREADS; j++) {
            es.execute(new MyThread(j));
        }
        es.shutdown();
        while (!es.isTerminated()) {
        }
        long endTime = System.currentTimeMillis();
        long timeDiff = endTime - startTime;
        System.out.println("Program Execution Time: " + timeDiff + "ms");
        System.out.println("1..." + (NUM_END - 1) + " prime# counter=" + counter);
    }

    private static boolean isPrime(int x) {
        if (x <= 1)
            return false;
        for (int i = 2; i < x; i++) {
            if (x % i == 0)
                return false;
        }
        return true;
    }

    private static int getNumber() {
        synchronized (lock) {
            return tmp++;
        }
    }

    public static class MyThread implements Runnable {
        private int id;

        public MyThread(int id) {
            this.id = id;
        }

        public void run() {
            long startTime = System.currentTimeMillis();
            int i;
            while ((i = getNumber()) < NUM_END) {
                if (isPrime(i))
                    synchronized (lock) {
                        counter++;
                    }
            }
            long endTime = System.currentTimeMillis();
            long timeDiff = endTime - startTime;
            System.out.println("Thread#" + id + " Execution Time: " + timeDiff +
"ms");  }
    }
}
```