Concurrent programming JAVA

Maxence Labourel 50241659

BlockingQueue and ArrayBlockingQueue....1
Example.... 1
Output :....1
Code :....2
ReadWriteLock....3
Example.... 3
Output :....3
Code :....4
AtomicInteger....5
Example.... 5
Output :....5
Code :....6
CyclicBarrier....7
Example.... 7
Output :....7
Code :....7
ExecutorService, Executors, Callable, Future....8
Example.... 8
Output :....8
Code :....9

BlockingQueue and ArrayBlockingQueue

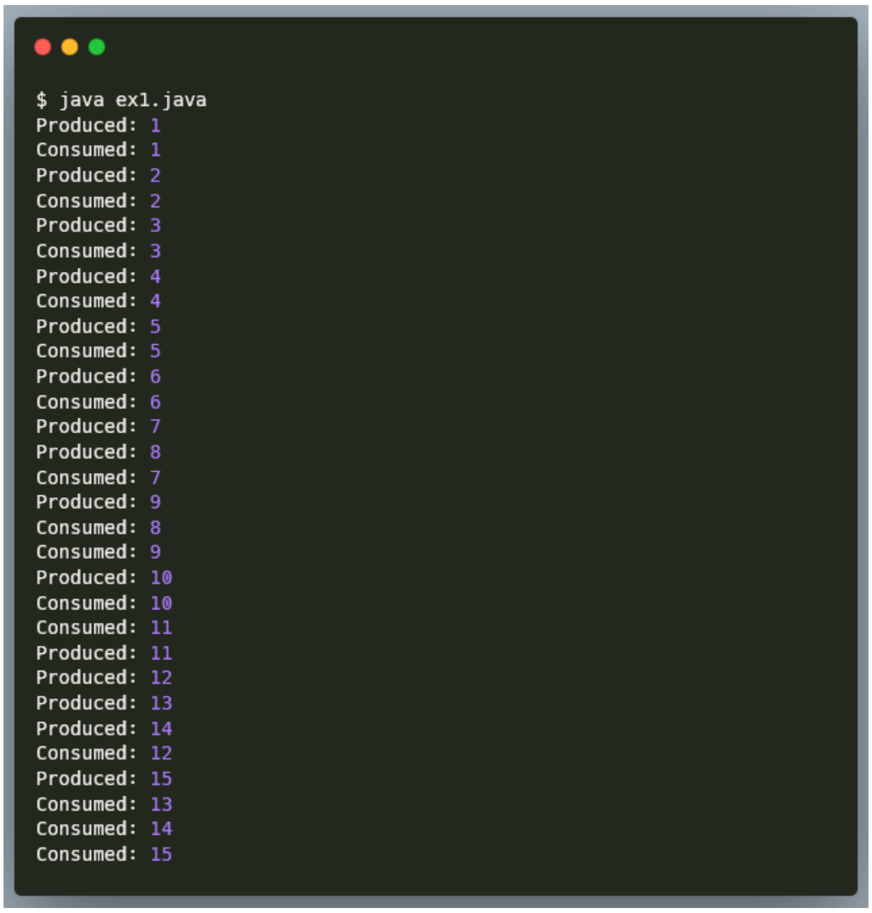
A **BlockingQueue** is an interface that represents a thread safe **Queue** (which means we can take and add elements in a concurrent algorithm without concurrency issues) of fixed size. It is called **Blocking** because it can make a thread wait if :

- The thread tries to take an element from the Queue and the Queue is empty, in that case the thread will be resumed when a new element is inserted in the Queue
- The thread tries to put an element into the Queue but the Queue is full, in that case the thread will be resumed once there is an empty space in the Queue

ArrayBlockingQueue is an implementation of BlockingQueue that uses an array. The elements are ordered in FIFO and **blocks** threads that try to **add elements** when the queue is **full** or **remove elements** when the queue is **empty**.

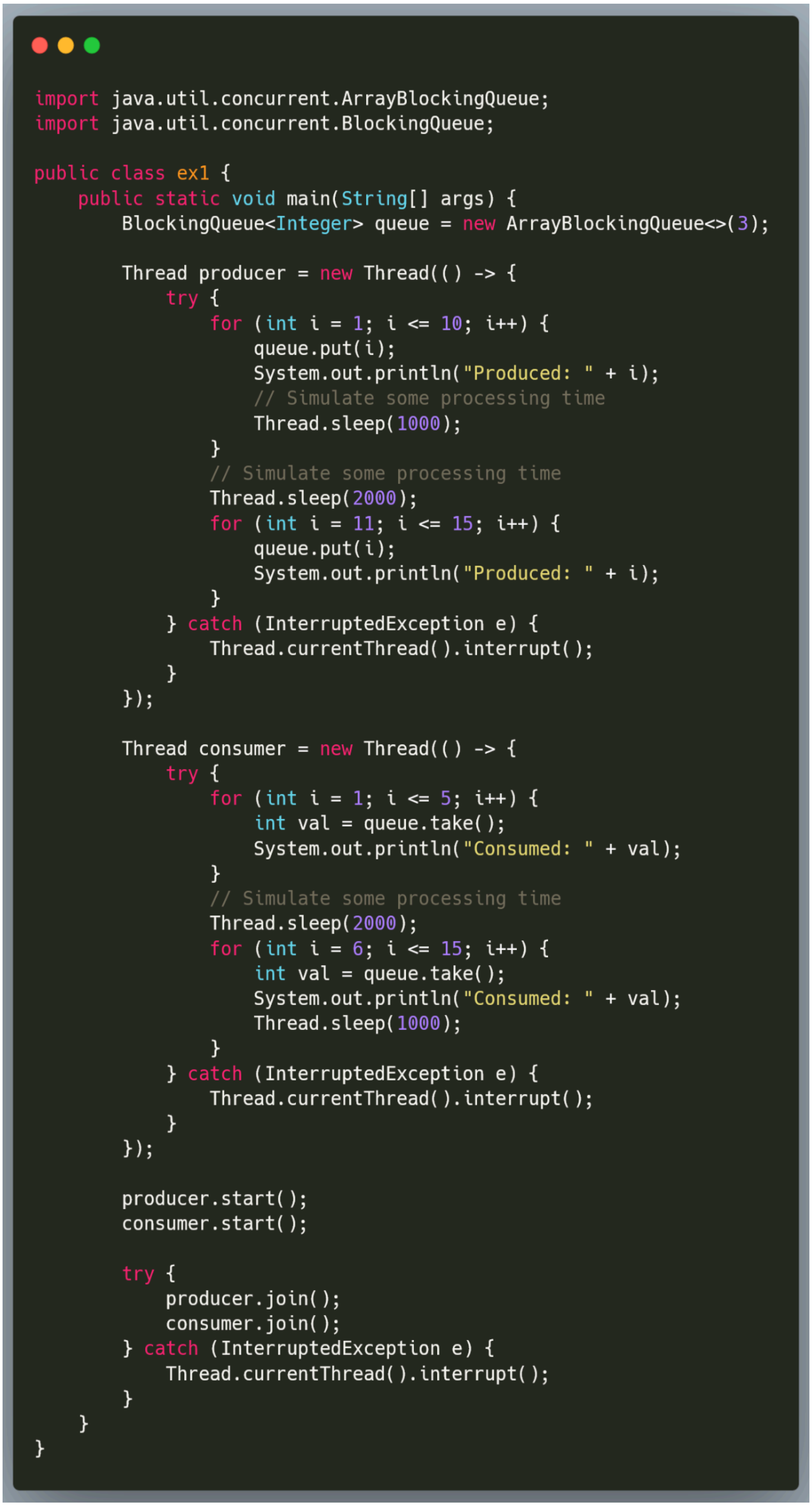
Example

Output :



```
$ java ex1.java
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
Produced: 6
Consumed: 6
Produced: 7
Produced: 8
Consumed: 7
Produced: 9
Consumed: 8
Consumed: 9
Produced: 10
Consumed: 10
Consumed: 11
Produced: 11
Produced: 12
Produced: 13
Produced: 14
Consumed: 12
Produced: 15
Consumed: 13
Consumed: 14
Consumed: 15
```

Code :



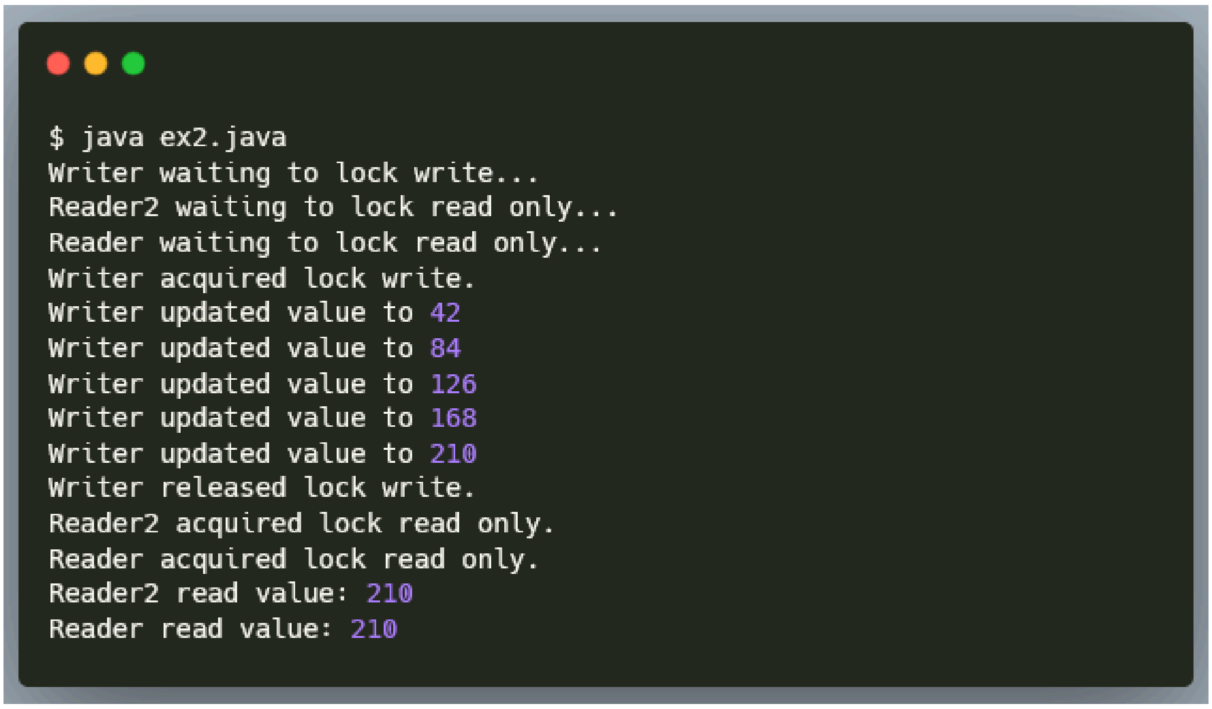
```java
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class ex1 {
    public static void main(String[] args) {
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(3);

        Thread producer = new Thread(() -> {
            try {
                for (int i = 1; i <= 10; i++) {
                    queue.put(i);
                    System.out.println("Produced: " + i);
                    // Simulate some processing time
                    Thread.sleep(1000);
                }
                // Simulate some processing time
                Thread.sleep(2000);
                for (int i = 11; i <= 15; i++) {
                    queue.put(i);
                    System.out.println("Produced: " + i);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        Thread consumer = new Thread(() -> {
            try {
                for (int i = 1; i <= 5; i++) {
                    int val = queue.take();
                    System.out.println("Consumed: " + val);
                }
                // Simulate some processing time
                Thread.sleep(2000);
                for (int i = 6; i <= 15; i++) {
                    int val = queue.take();
                    System.out.println("Consumed: " + val);
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        producer.start();
        consumer.start();

        try {
            producer.join();
            consumer.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

ReadWriteLock

ReadWriteLock is an interface that **holds a pair of associated locks**, one for read-only operations and one for write operations. The **read** lock can be held by an **infinite amount of reader threads**, as long as there are no writers, meanwhile the **write** lock is **exclusive**.

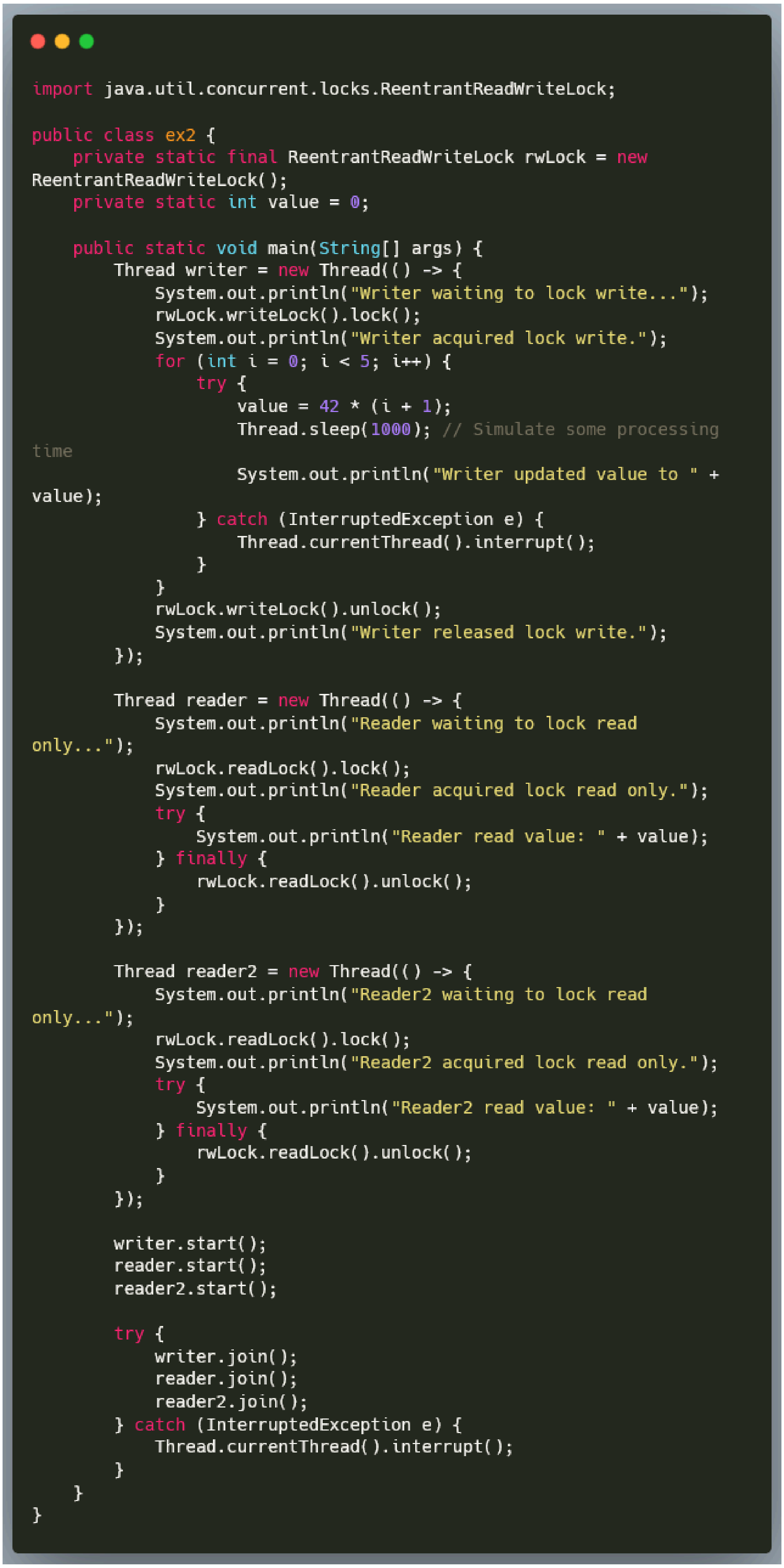
Example

Output :



```
$ java ex2.java
Writer waiting to lock write...
Reader2 waiting to lock read only...
Reader waiting to lock read only...
Writer acquired lock write.
Writer updated value to 42
Writer updated value to 84
Writer updated value to 126
Writer updated value to 168
Writer updated value to 210
Writer released lock write.
Reader2 acquired lock read only.
Reader acquired lock read only.
Reader2 read value: 210
Reader read value: 210
```

Code :



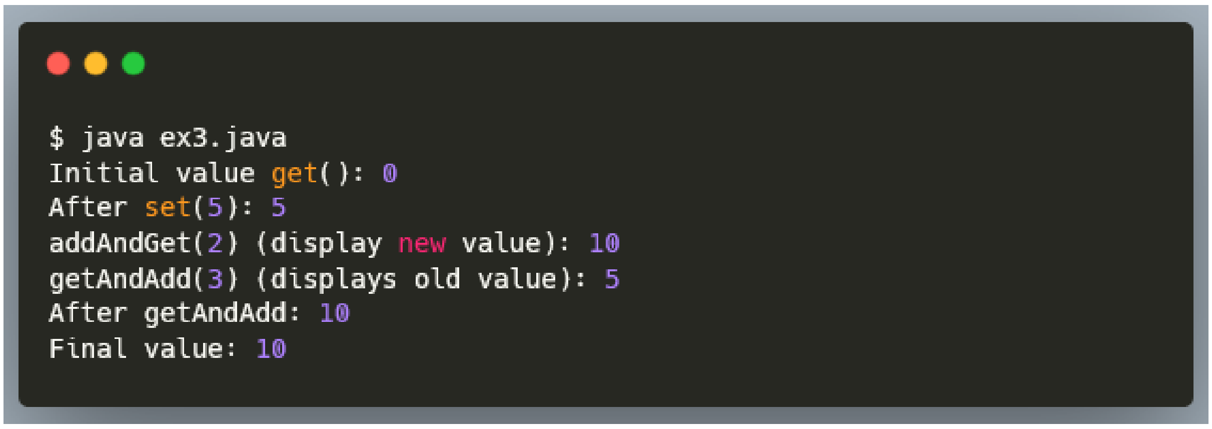
```
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class ex2 {
    private static final ReentrantReadWriteLock rwLock = new
ReentrantReadWriteLock();
    private static int value = 0;

    public static void main(String[] args) {
        Thread writer = new Thread(() -> {
            System.out.println("Writer waiting to lock write...");
            rwLock.writeLock().lock();
            System.out.println("Writer acquired lock write.");
            for (int i = 0; i < 5; i++) {
                try {
                    value = 42 * (i + 1);
                    Thread.sleep(1000); // Simulate some processing
time
                    System.out.println("Writer updated value to " +
value);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
            rwLock.writeLock().unlock();
            System.out.println("Writer released lock write.");
        });

        Thread reader = new Thread(() -> {
            System.out.println("Reader waiting to lock read
only...");
            rwLock.readLock().lock();
            System.out.println("Reader acquired lock read only.");
            try {
                System.out.println("Reader read value: " + value);
            } finally {
                rwLock.readLock().unlock();
            }
        });

        Thread reader2 = new Thread(() -> {
            System.out.println("Reader2 waiting to lock read
only...");
            rwLock.readLock().lock();
            System.out.println("Reader2 acquired lock read only.");
            try {
                System.out.println("Reader2 read value: " + value);
            } finally {
                rwLock.readLock().unlock();
            }
        });

        writer.start();
        reader.start();
        reader2.start();

        try {
            writer.join();
            reader.join();
            reader2.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

AtomicInteger

AtomicInteger is a class that provides **atomic (thread safe) operations** on integers. It allows **lock-free, thread-safe operations but on a single variable**. Which means there are **no critical parts in the code allowing for very quick operations**. You can increment, decrement and update the value atomically.

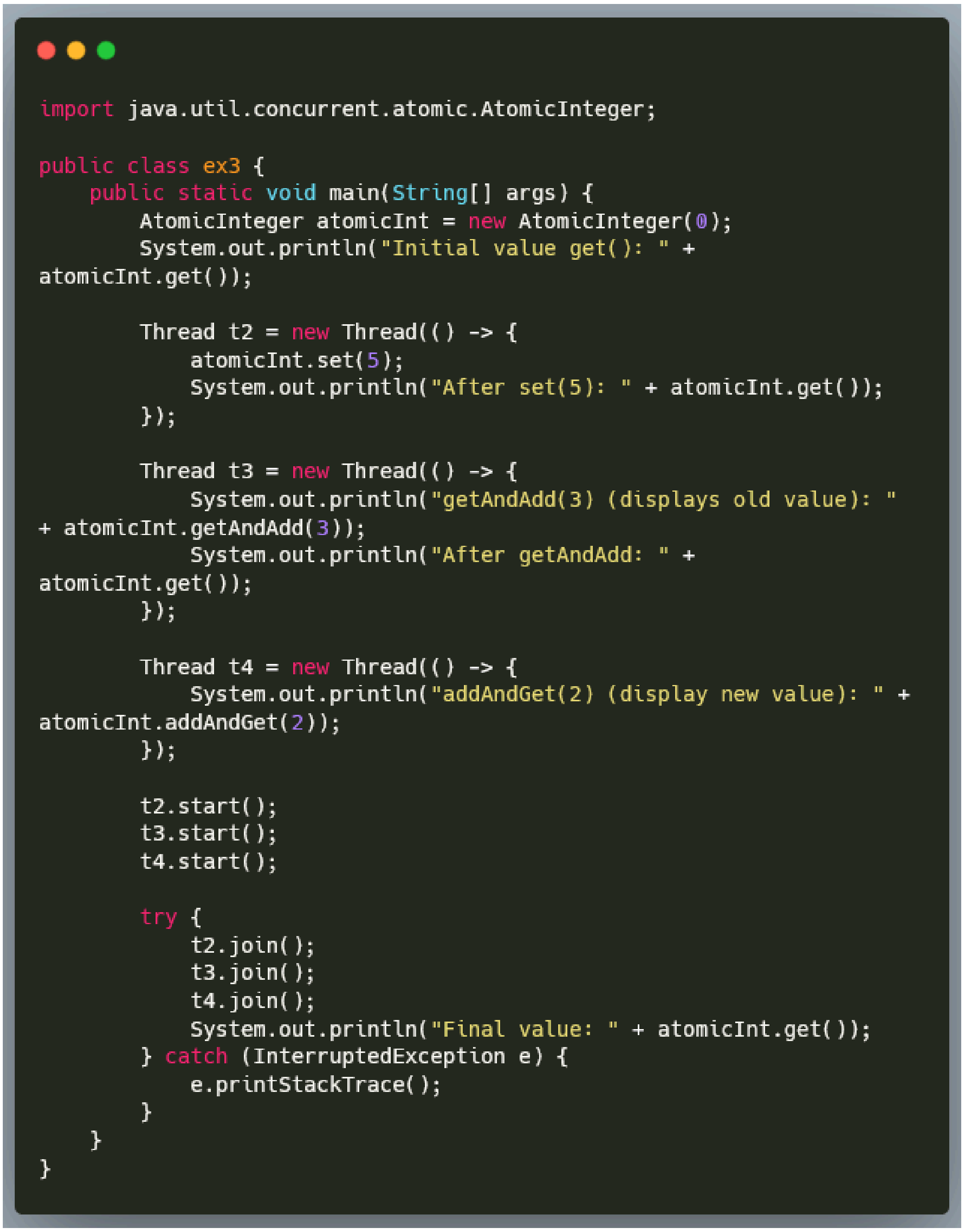
Example

Output :



```
$ java ex3.java
Initial value get(): 0
After set(5): 5
addAndGet(2) (display new value): 10
getAndAdd(3) (displays old value): 5
After getAndAdd: 10
Final value: 10
```

Code :



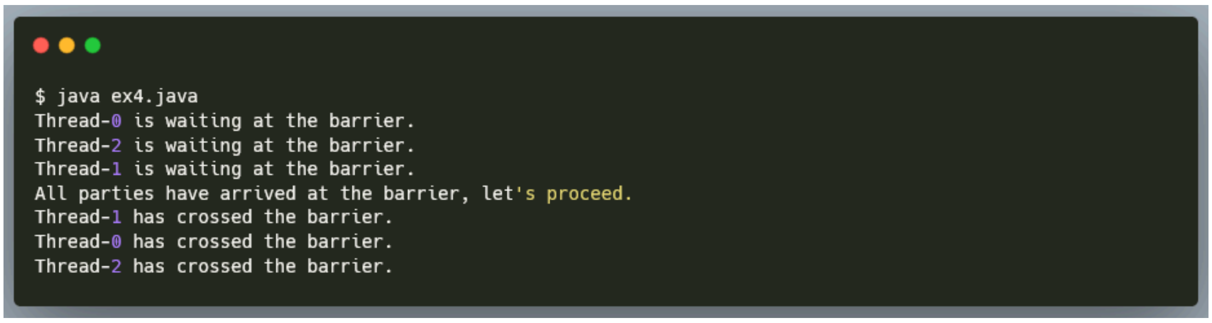
```java
import java.util.concurrent.atomic.AtomicInteger;

public class ex3 {
    public static void main(String[] args) {
        AtomicInteger atomicInt = new AtomicInteger(0);
        System.out.println("Initial value get(): " +
atomicInt.get());

        Thread t2 = new Thread(() -> {
            atomicInt.set(5);
            System.out.println("After set(5): " + atomicInt.get());
        });

        Thread t3 = new Thread(() -> {
            System.out.println("getAndAdd(3) (displays old value): "
+ atomicInt.getAndAdd(3));
            System.out.println("After getAndAdd: " +
atomicInt.get());
        });

        Thread t4 = new Thread(() -> {
            System.out.println("addAndGet(2) (display new value): " +
atomicInt.addAndGet(2));
        });

        t2.start();
        t3.start();
        t4.start();

        try {
            t2.join();
            t3.join();
            t4.join();
            System.out.println("Final value: " + atomicInt.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

CyclicBarrier

CyclicBarrier is a **synchronisation** type that allows **multiple threads to all wait for each other** to reach a common barrier. It is useful when threads must wait for each others to reach a certain point before proceeding.

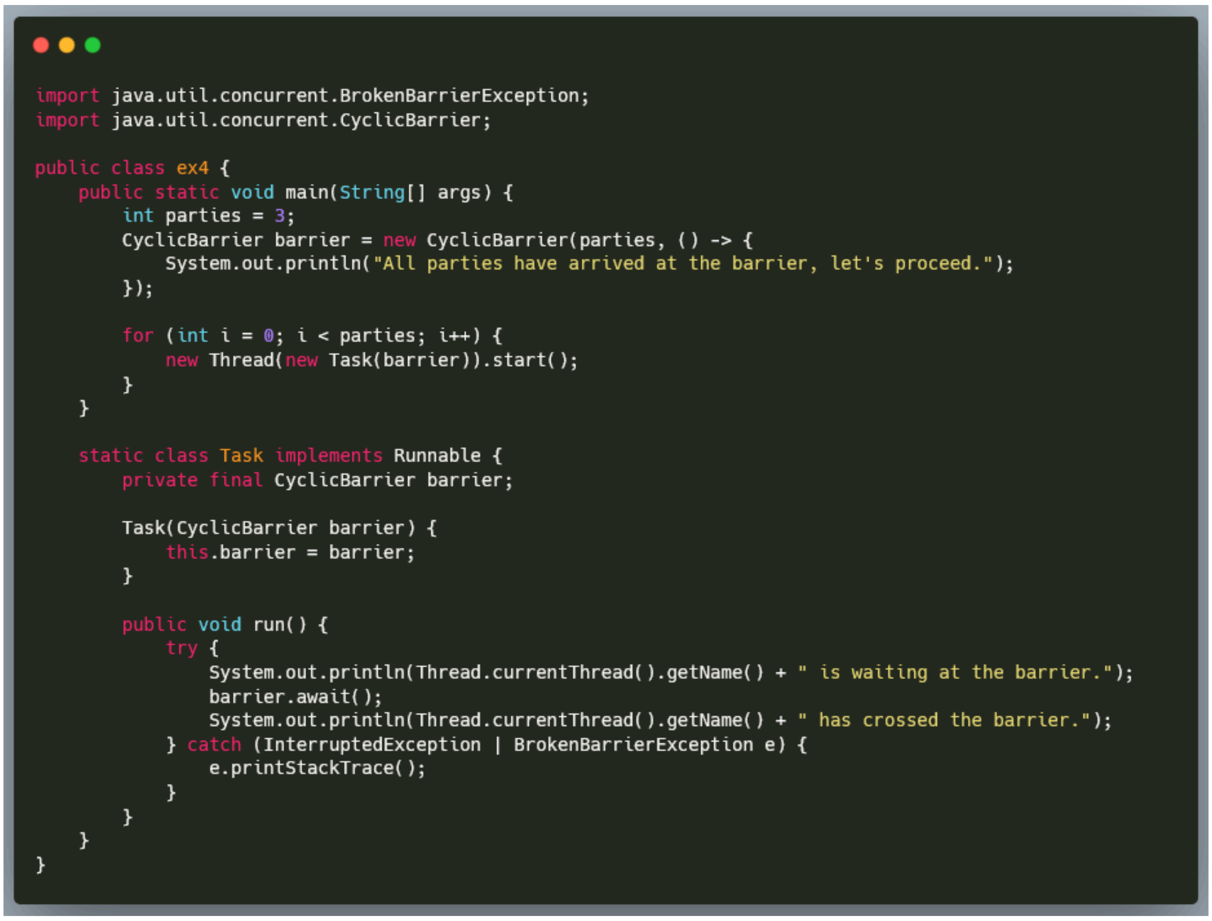
Example

Output :



```
$ java ex4.java
Thread-0 is waiting at the barrier.
Thread-2 is waiting at the barrier.
Thread-1 is waiting at the barrier.
All parties have arrived at the barrier, let's proceed.
Thread-1 has crossed the barrier.
Thread-0 has crossed the barrier.
Thread-2 has crossed the barrier.
```

Code :



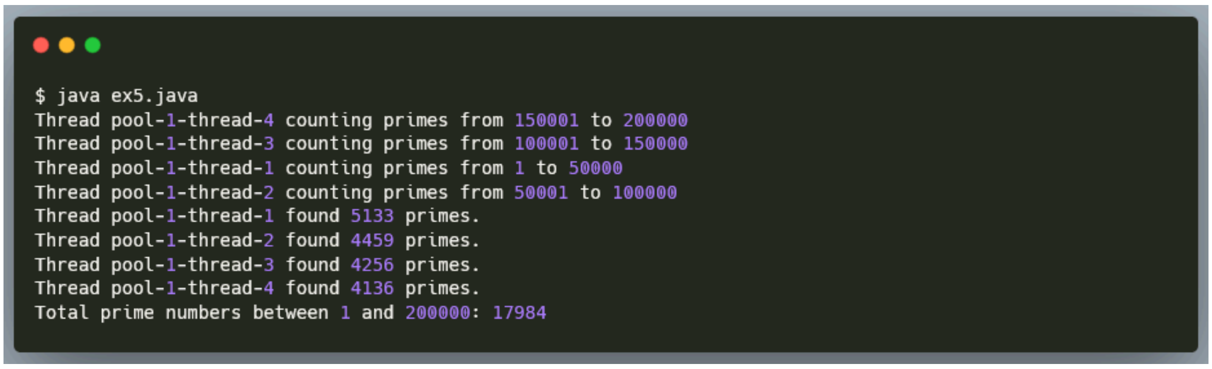
```java
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class ex4 {
    public static void main(String[] args) {
        int parties = 3;
        CyclicBarrier barrier = new CyclicBarrier(parties, () -> {
            System.out.println("All parties have arrived at the barrier, let's proceed.");
        });

        for (int i = 0; i < parties; i++) {
            new Thread(new Task(barrier)).start();
        }
    }

    static class Task implements Runnable {
        private final CyclicBarrier barrier;

        Task(CyclicBarrier barrier) {
            this.barrier = barrier;
        }

        public void run() {
            try {
                System.out.println(Thread.currentThread().getName() + " is waiting at the barrier.");
                barrier.await();
                System.out.println(Thread.currentThread().getName() + " has crossed the barrier.");
            } catch (InterruptedException | BrokenBarrierException e) {
                e.printStackTrace();
            }
        }
    }
}
```

ExecutorService, Executors, Callable, Future

- **ExecutorService** is an interface that provides methods to **manage termination** and methods that can **produce a Future** in order to track the progress of one or more **asynchronous tasks**.
- **Executors** is a **Factory** class that provides methods to construct **different executor services**.
- **Callable** is similar to **Runnable** but **returns a result and can throw an exception**.
- **Future** represents the **result of an asynchronous task**.

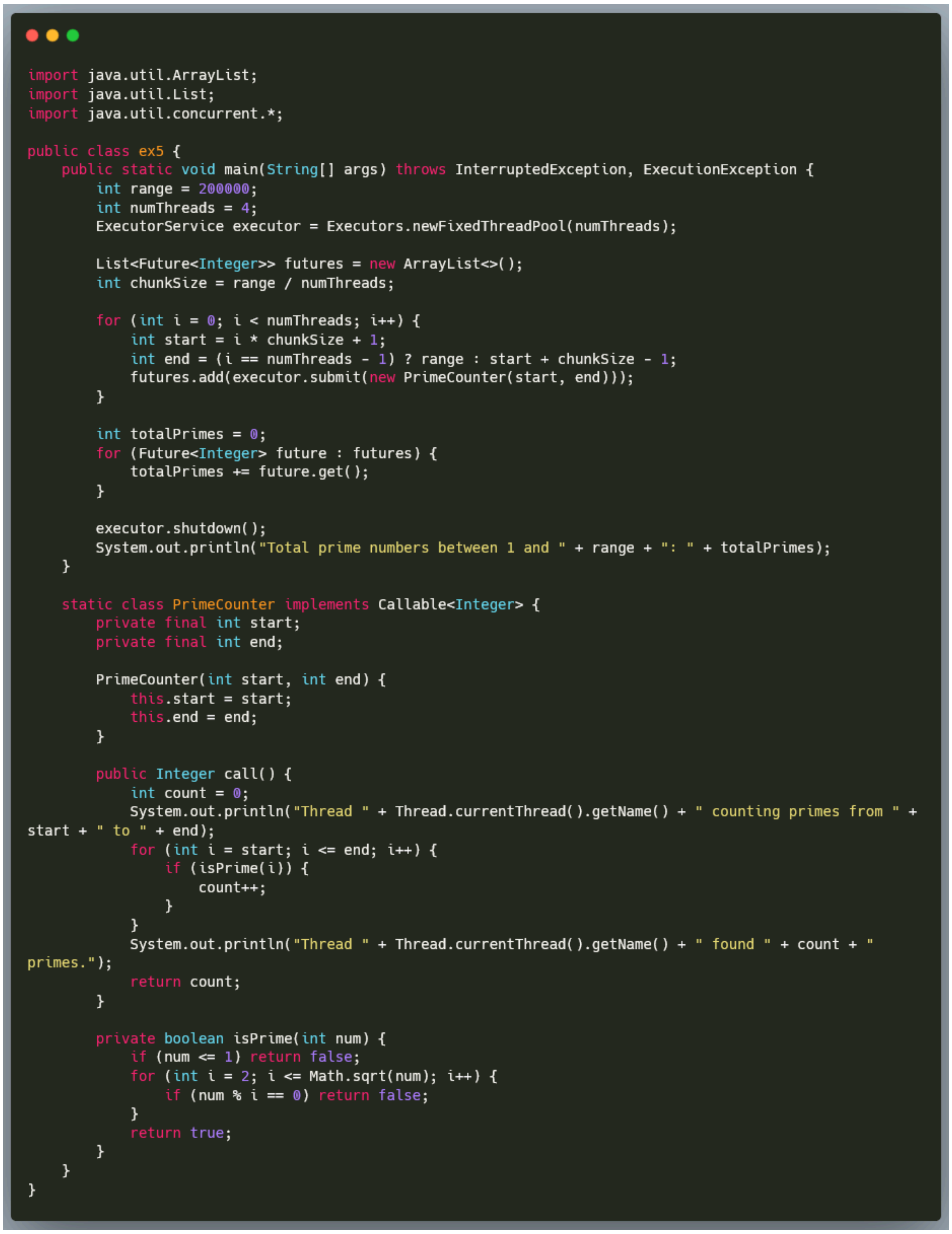
Example

Output :



```
$ java ex5.java
Thread pool-1-thread-4 counting primes from 150001 to 200000
Thread pool-1-thread-3 counting primes from 100001 to 150000
Thread pool-1-thread-1 counting primes from 1 to 50000
Thread pool-1-thread-2 counting primes from 50001 to 100000
Thread pool-1-thread-1 found 5133 primes.
Thread pool-1-thread-2 found 4459 primes.
Thread pool-1-thread-3 found 4256 primes.
Thread pool-1-thread-4 found 4136 primes.
Total prime numbers between 1 and 200000: 17984
```

Code :



```java
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

public class ex5 {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        int range = 200000;
        int numThreads = 4;
        ExecutorService executor = Executors.newFixedThreadPool(numThreads);

        List<Future<Integer>> futures = new ArrayList<>();
        int chunkSize = range / numThreads;

        for (int i = 0; i < numThreads; i++) {
            int start = i * chunkSize + 1;
            int end = (i == numThreads - 1) ? range : start + chunkSize - 1;
            futures.add(executor.submit(new PrimeCounter(start, end)));
        }

        int totalPrimes = 0;
        for (Future<Integer> future : futures) {
            totalPrimes += future.get();
        }

        executor.shutdown();
        System.out.println("Total prime numbers between 1 and " + range + ": " + totalPrimes);
    }

    static class PrimeCounter implements Callable<Integer> {
        private final int start;
        private final int end;

        PrimeCounter(int start, int end) {
            this.start = start;
            this.end = end;
        }

        public Integer call() {
            int count = 0;
            System.out.println("Thread " + Thread.currentThread().getName() + " counting primes from " +
start + " to " + end);
            for (int i = start; i <= end; i++) {
                if (isPrime(i)) {
                    count++;
                }
            }
            System.out.println("Thread " + Thread.currentThread().getName() + " found " + count + "
primes.");
            return count;
        }

        private boolean isPrime(int num) {
            if (num <= 1) return false;
            for (int i = 2; i <= Math.sqrt(num); i++) {
                if (num % i == 0) return false;
            }
            return true;
        }
    }
}
```