Project #3

과목명	멀티코어 컴퓨팅
담당교수	손봉수 교수님
학과	소프트웨어학과
학번	20184806
이름	이상은
제출일	2020. 05. 17

Environment

CPU : Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz   2.90 GHz

OS : Windows 10 Pro

Number Of Cores : 6

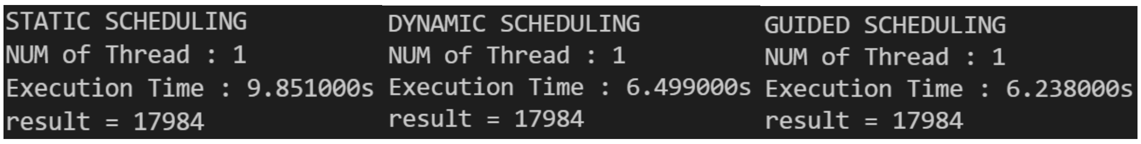
Memory Size : RAM 16.0GB

GitHub : https://github.com/HaruToy/MulticoreComputing.git

Problem 1. Count Prime Number

The problem of obtaining a prime number up to N

Result

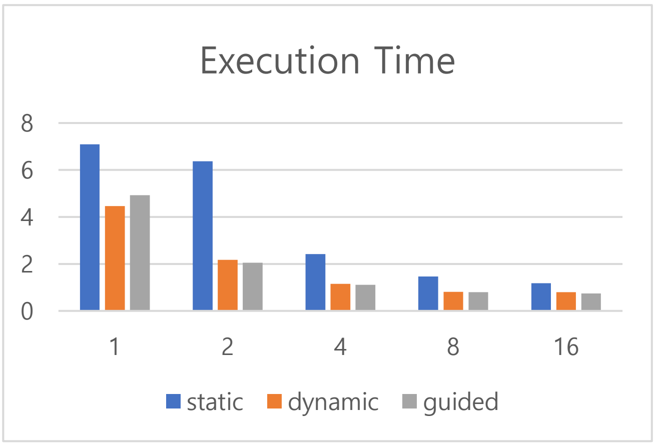


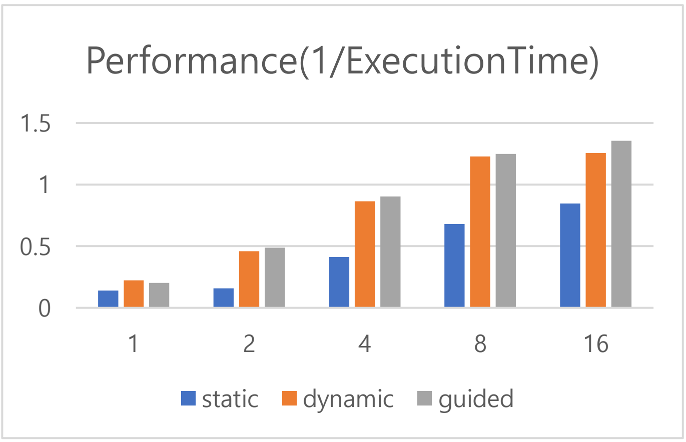
```
STATIC SCHEDULING              DYNAMIC SCHEDULING             GUIDED SCHEDULING
NUM of Thread : 1              NUM of Thread : 1              NUM of Thread : 1
Execution Time : 9.851000s     Execution Time : 6.499000s     Execution Time : 6.238000s
result = 17984                 result = 17984                 result = 17984
```

[image 1] Result of counting Prime Number

The number of prime numbers up to 20000 is 17984.

Performance





[graph 1] Execution Time and Performance per number of threads

exec time	1	2	4	8	16
static	7.091	6.372	2.42	1.469	1.18
dynamic	4.462	2.177	1.155	0.814	0.796
guided	4.925	2.053	1.105	0.8	0.737

(단, 소수점 아래 넷째 자리에서 반올림)

performance	1	2	4	8	16
static	0.141024	0.156937	0.413223	0.680735	0.847458
dynamic	0.224115	0.459348	0.865801	1.228501	1.256281
guided	0.203046	0.487092	0.904977	1.25	1.356852

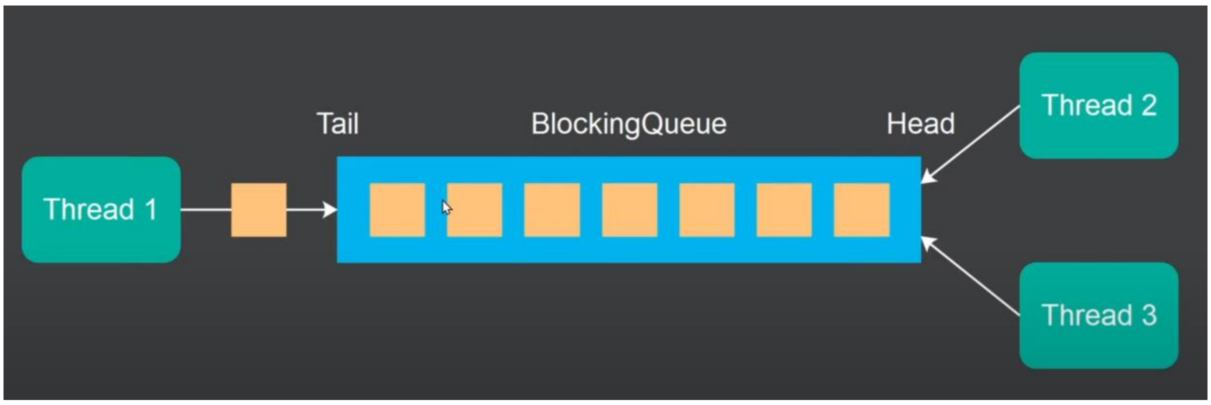
(단, 소수점 아래 일곱 번째 자리에서 반올림)

[table 2] Execution Time and Performance per number of threads

Overall, the larger the number of Threads, the shorter the execution time. The number of cores on my computer was six, so there was no significant difference between eight and sixteen threads. In the case of two threads, the "Static" method for large numbers was less efficient than the dynamic or guided method due to its limitations in equitable distribution. It takes longer to know the large number is a prime number or not.

Problem 2. Semaphore

1-1. BlockingQueue & ArrayBlockingQueue

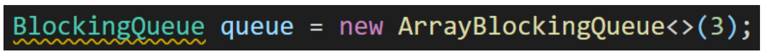


[image 2] Blocking Queue's Figure

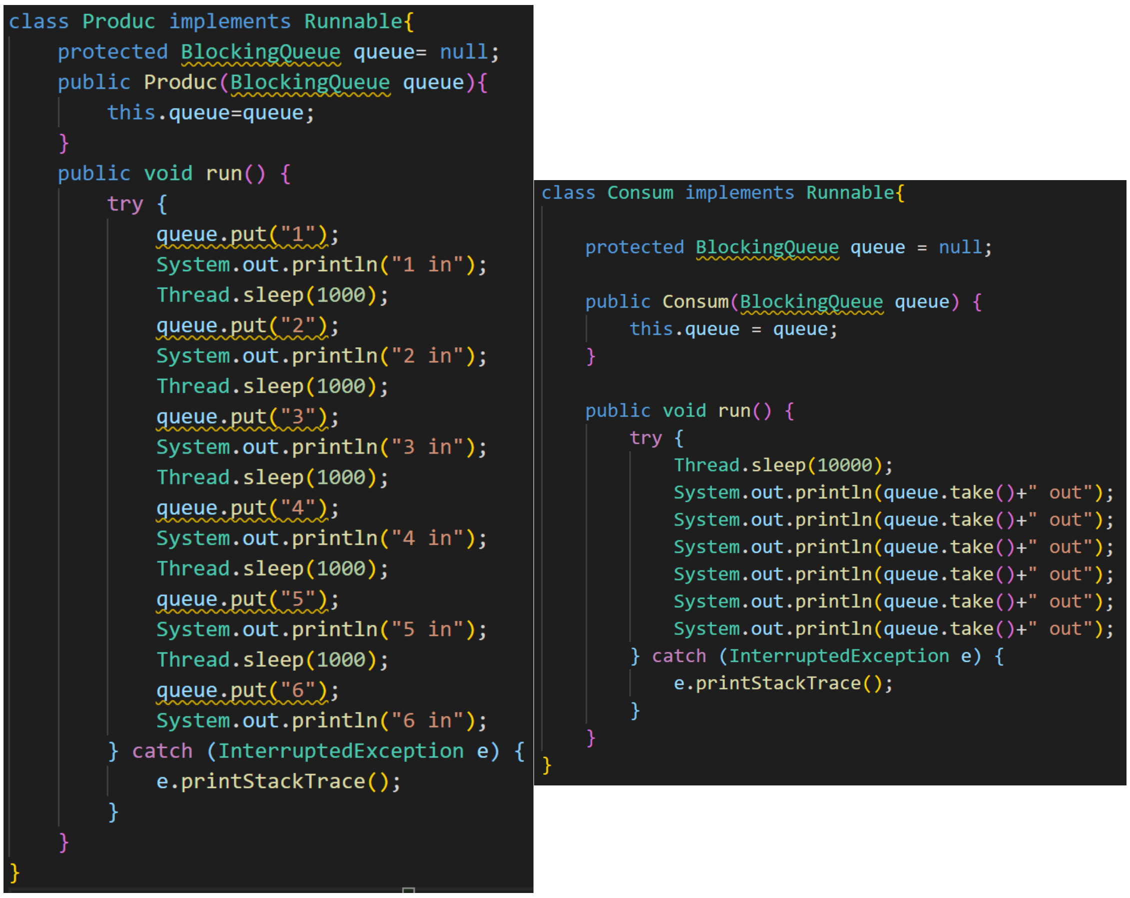
Blocking Queue makes thread to put elements or pull elements safely. If there is no element in the queue, the 'pull' command is blocked until any element to be 'pushed'. And vice versa.

Array Blocking Queue is similar to Blocking Queue. It also makes thread to put elements or pull elements safely. Unlike Blocking Queue, Array Blocking Queue must have a limited size of the queue.

1-2. Example Code of Blocking Queue



```
BlockingQueue queue = new ArrayBlockingQueue<>(3);
```

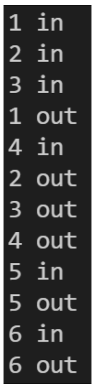


```
class Produc implements Runnable{
    protected BlockingQueue queue= null;
    public Produc(BlockingQueue queue){
        this.queue=queue;
    }
    public void run() {
        try {
            queue.put("1");
            System.out.println("1 in");
            Thread.sleep(1000);
            queue.put("2");
            System.out.println("2 in");
            Thread.sleep(1000);
            queue.put("3");
            System.out.println("3 in");
            Thread.sleep(1000);
            queue.put("4");
            System.out.println("4 in");
            Thread.sleep(1000);
            queue.put("5");
            System.out.println("5 in");
            Thread.sleep(1000);
            queue.put("6");
            System.out.println("6 in");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
class Consum implements Runnable{

    protected BlockingQueue queue = null;

    public Consum(BlockingQueue queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            Thread.sleep(10000);
            System.out.println(queue.take()+" out");
            System.out.println(queue.take()+" out");
            System.out.println(queue.take()+" out");
            System.out.println(queue.take()+" out");
            System.out.println(queue.take()+" out");
            System.out.println(queue.take()+" out");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

[image 3] Producer Thread and Consumer Thread of example code

I declared Array Blocking Queue which size is 3, 'Produc' Thread puts number into the Queue and 'Consum' Thread takes number from the Queue. And Run 'Produc' Thread first. After putting number "3", 'Produc' Thread is blocked until 'Consum' Thread takes an element from the Queue. After taking "4", 'Consum' Thread is blocked until 'Produc' Thread puts an element form the Queue.



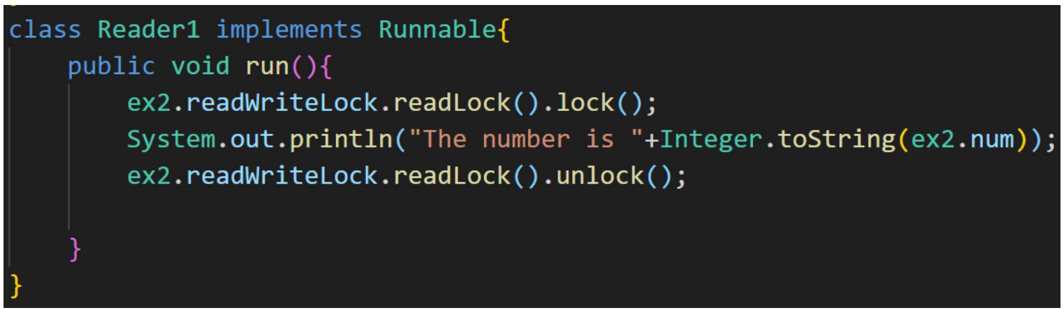
```
1 in
2 in
3 in
1 out
4 in
2 out
3 out
4 out
5 in
5 out
6 in
6 out
```

[image 4] Result of example code

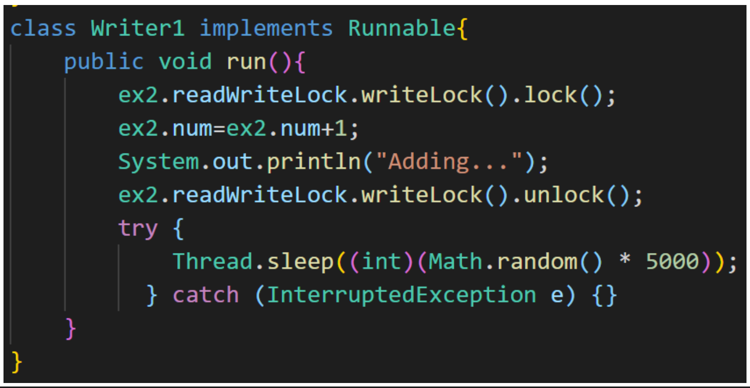
2-1. ReadWriteLock

Read Write Lock allows to lock shared variable. When other threads don't use write lock, you can use read lock to read the variable safely. When other threads don't use read lock and write lock, you can use write lock to write the shared variable safely. Before you read a shared variable, Use "readLock().lock();" to read variable. After you read, Use "readLock.unlock()" to make other threads modify the variable. The case of writing is similar to it.

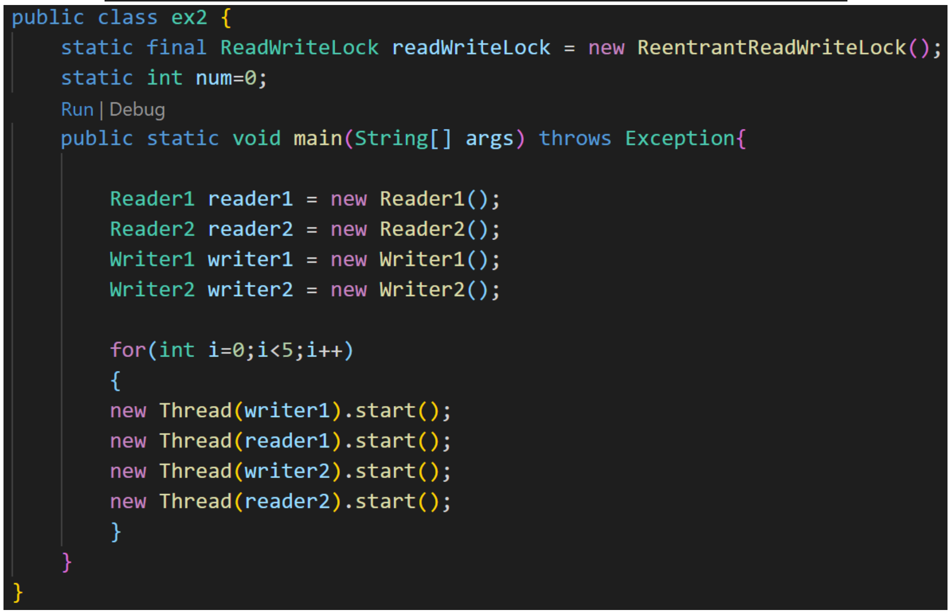
2-2. Example Code of ReadWriteLock



```java
class Reader1 implements Runnable{
    public void run(){
        ex2.readWriteLock.readLock().lock();
        System.out.println("The number is "+Integer.toString(ex2.num));
        ex2.readWriteLock.readLock().unlock();

    }
}
```



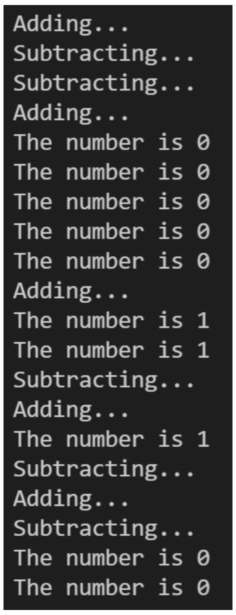
```java
class Writer1 implements Runnable{
    public void run(){
        ex2.readWriteLock.writeLock().lock();
        ex2.num=ex2.num+1;
        System.out.println("Adding...");
        ex2.readWriteLock.writeLock().unlock();
        try {
            Thread.sleep((int)(Math.random() * 5000));
        } catch (InterruptedException e) {}
    }
}
```



```java
public class ex2 {
    static final ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
    static int num=0;
    public static void main(String[] args) throws Exception{

        Reader1 reader1 = new Reader1();
        Reader2 reader2 = new Reader2();
        Writer1 writer1 = new Writer1();
        Writer2 writer2 = new Writer2();

        for(int i=0;i<5;i++)
        {
        new Thread(writer1).start();
        new Thread(reader1).start();
        new Thread(writer2).start();
        new Thread(reader2).start();
        }
    }
}
```

[image 5] Example of Readlock and Writelock

"Reader1" Thread and "Reader2" Thread are the same thread. They read the "num" which is a shared variable. Writer1 Thread adds 1 to "num". Writer2 Thread subtracts 1 from "num". They use write lock to modify the shared variable safely. By using random sleep function, various threads are randomized.



```
Adding...
Subtracting...
Subtracting...
Adding...
The number is 0
The number is 0
The number is 0
The number is 0
The number is 0
Adding...
The number is 1
The number is 1
Subtracting...
Adding...
The number is 1
Subtracting...
Adding...
Subtracting...
The number is 0
The number is 0
```

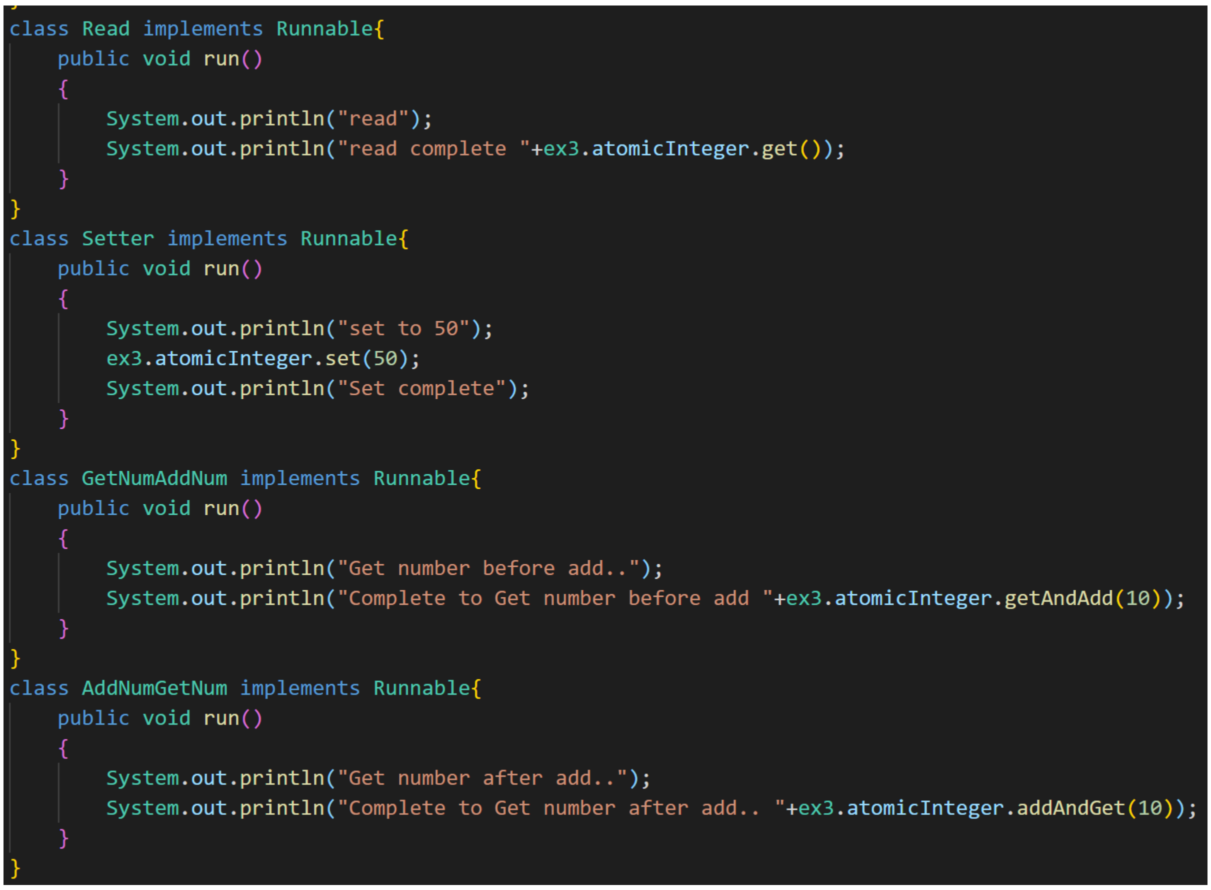
[image 6] Result of Example Code

The Initial value of "num" is 0. After adding 1 to 0 twice and subtracting 1 from 0 twice, the value of "num" becomes 0. Then add 1 to num once. The value of "num" becomes 1. Since then, results have been printed well according to the thread's execution.

3-1. Atomic Integer

Atomic integer is a safe variable of integer type. By using an atomic integer, we can read and write the shared variable safely. It provides method, “get()”, “set()”. And these guarantee the safe reading and writing.

3-2. Example Code

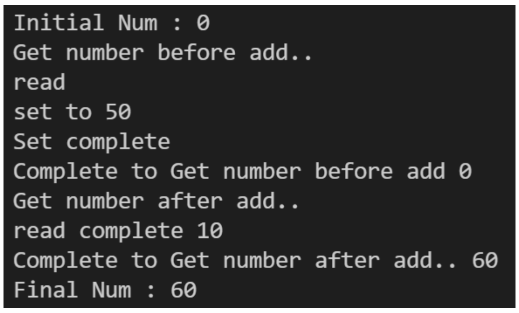


```java
class Read implements Runnable{
    public void run()
    {
        System.out.println("read");
        System.out.println("read complete "+ex3.atomicInteger.get());
    }
}
class Setter implements Runnable{
    public void run()
    {
        System.out.println("set to 50");
        ex3.atomicInteger.set(50);
        System.out.println("Set complete");
    }
}
class GetNumAddNum implements Runnable{
    public void run()
    {
        System.out.println("Get number before add..");
        System.out.println("Complete to Get number before add "+ex3.atomicInteger.getAndAdd(10));
    }
}
class AddNumGetNum implements Runnable{
    public void run()
    {
        System.out.println("Get number after add..");
        System.out.println("Complete to Get number after add.. "+ex3.atomicInteger.addAndGet(10));
    }
}
```

[image 7] Example of atomic integer

Read Thread reads the value of atomic integer. Setter Thread sets the value of atomic integer to 50. GetNumAddNum Thread gets the value of atomic integer and then adds 10 to the value of atomic integer. AddNumGetNum Thread adds 10 to the value of atomic integer and then gets the value of atomic integer.

Because of the print delay, I print which thread is executing. And after executing the thread, I print the result of thread.



```
Initial Num : 0
Get number before add..
read
set to 50
Set complete
Complete to Get number before add 0
Get number after add..
read complete 10
Complete to Get number after add.. 60
Final Num : 60
```

[image 8] Result of Example

Thread Execution Order

1. Get number and then add 10 to it.

   ⇨ Complete to Get number before add **0**

2. Read the number.

   ⇨ Read complete **10**

3. Set the number to 50.

   ⇨ Set to **50**

4. Add 10 to the number and then get the number.

   ⇨ Complete to Get number after add.. **60**

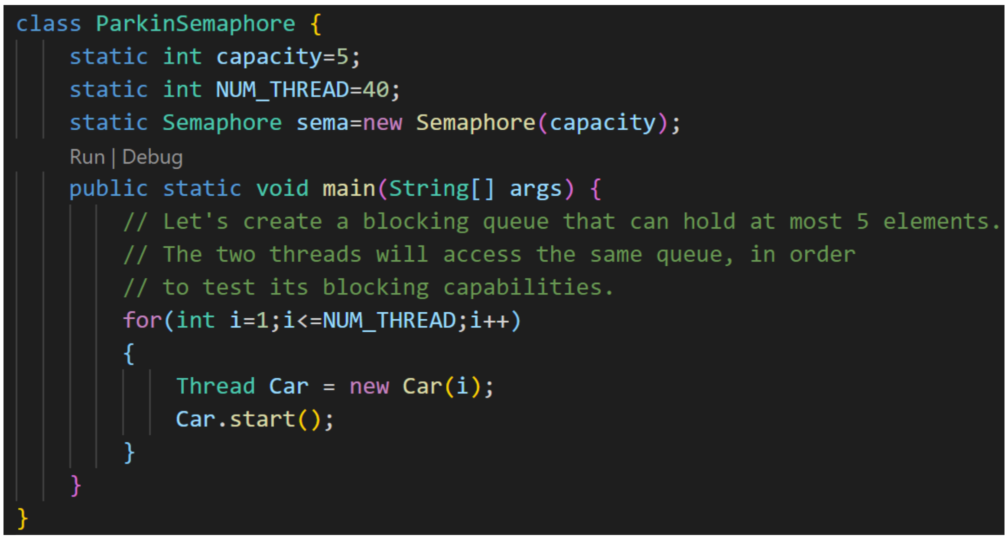
Finally, the result is **60**.

Threads were carried out simultaneously, but results were produced precisely in the order in which the threads started.

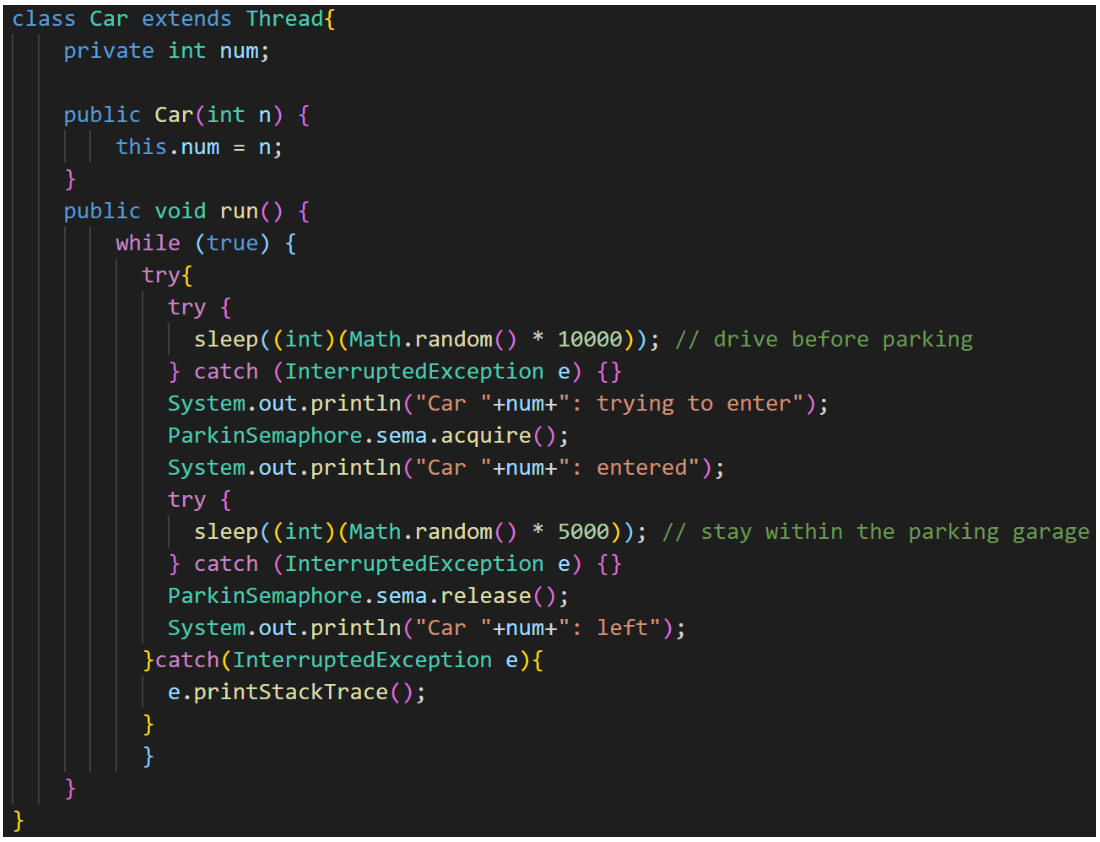
4-1. Semaphore

By using Semaphore, we can control the number of threads which enter the critical section. “acquire()” method permits a thread to enter the critical section. “release()” method permits a thread to return. If there are more threads to enter than I’ve specified, they wait until any threads return.

4-2. Example Code

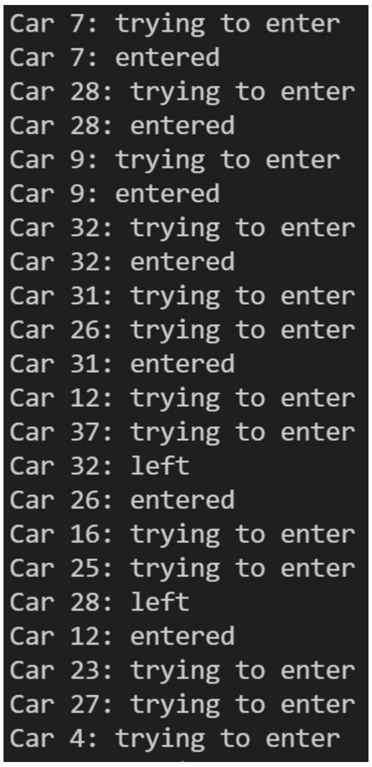


```java
class ParkinSemaphore {
    static int capacity=5;
    static int NUM_THREAD=40;
    static Semaphore sema=new Semaphore(capacity);
    Run | Debug
    public static void main(String[] args) {
        // Let's create a blocking queue that can hold at most 5 elements.
        // The two threads will access the same queue, in order
        // to test its blocking capabilities.
        for(int i=1;i<=NUM_THREAD;i++)
        {
            Thread Car = new Car(i);
            Car.start();
        }
    }
}
```



```java
class Car extends Thread{
    private int num;

    public Car(int n) {
        this.num = n;
    }
    public void run() {
        while (true) {
          try{
            try {
              sleep((int)(Math.random() * 10000)); // drive before parking
            } catch (InterruptedException e) {}
            System.out.println("Car "+num+": trying to enter");
            ParkinSemaphore.sema.acquire();
            System.out.println("Car "+num+": entered");
            try {
              sleep((int)(Math.random() * 5000)); // stay within the parking garage
            } catch (InterruptedException e) {}
            ParkinSemaphore.sema.release();
            System.out.println("Car "+num+": left");
          }catch(InterruptedException e){
            e.printStackTrace();
          }
        }
    }
}
```

[image 9] Example of Semaphore



```
Car 7: trying to enter
Car 7: entered
Car 28: trying to enter
Car 28: entered
Car 9: trying to enter
Car 9: entered
Car 32: trying to enter
Car 32: entered
Car 31: trying to enter
Car 26: trying to enter
Car 31: entered
Car 12: trying to enter
Car 37: trying to enter
Car 32: left
Car 26: entered
Car 16: trying to enter
Car 25: trying to enter
Car 28: left
Car 12: entered
Car 23: trying to enter
Car 27: trying to enter
Car 4: trying to enter
```

[image 10] Result of Parking Semaphore

Suppose we have a parking lot for only five cars. Define the number(5) of threads that are acceptable. Order Semaphore to acquire before the car entered the parking lot. Order Semaphore to release after the car left the parking lot.

As a result, only five vehicles entered the parking lot, and the incoming vehicle waited for the vehicle to leave the parking lot.