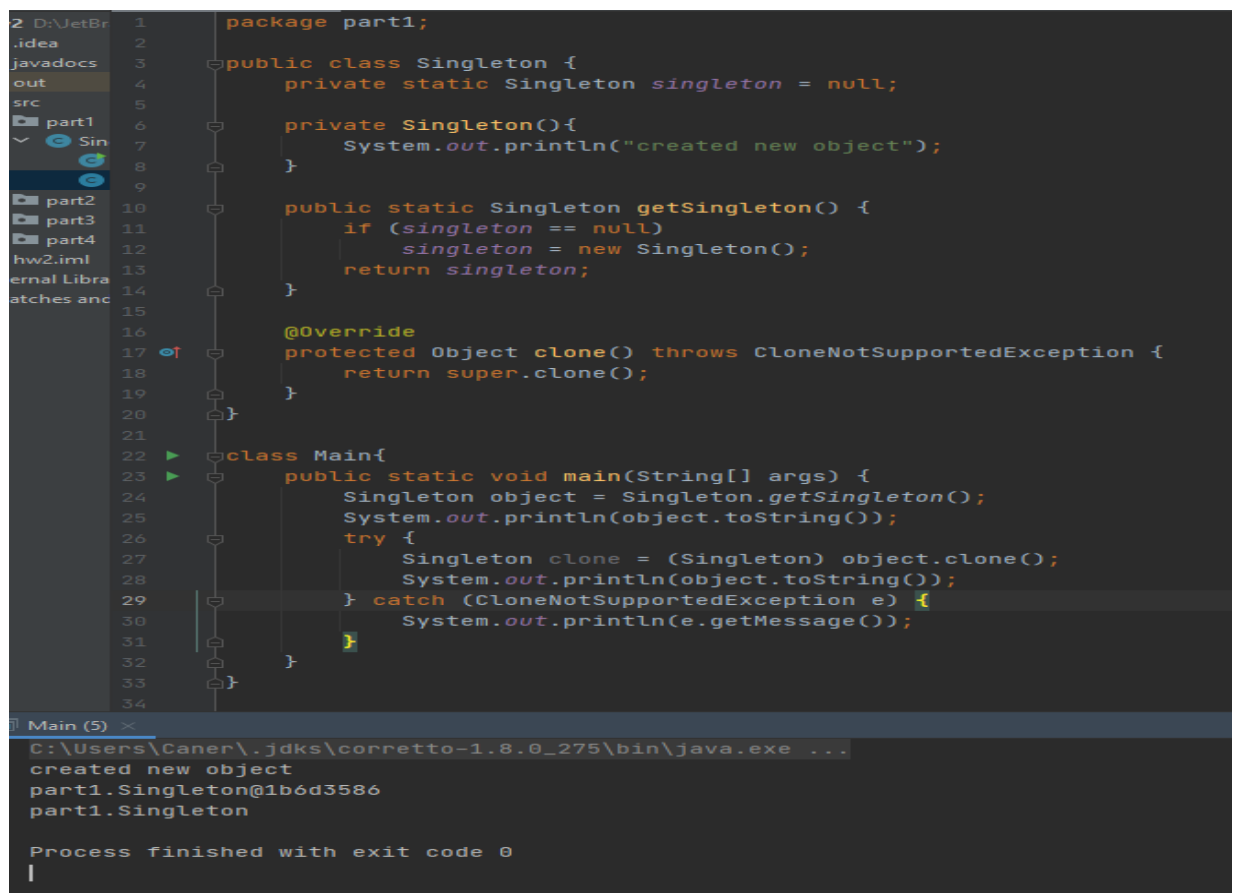


**CSE 443**  
**Object Oriented Analysis and Design**  
**Fall 2020– 2021**  
**HW2 Report**

**Caner KARAKAŞ**  
**131044061**

## Part 1

1. Singleton pattern is a software design pattern that allows the creation of only one object from a class type. This concept can sometimes be further developed and generalized to create a certain number of objects of the class. For example, the number of objects to be created can be limited to five. This is useful in situations where only a single object is required to coordinate behavior across the entire system. Some people consider this design pattern to be anti-mold, that it is used too much, and in some cases it is an unnecessary constraint to create the object one time. In this design pattern, it is possible to clone the object using the clone() method inherited from the Object class. However, since this design pattern is used according to a single object understanding, it is not a behavior suitable for the design pattern. However, the class that will use this design pattern for its construction must be implemented from the Clonable interface. In addition, the object class's clone method is called super.clone(). The Clone method does not create a new object, it can only duplicate an existing object.



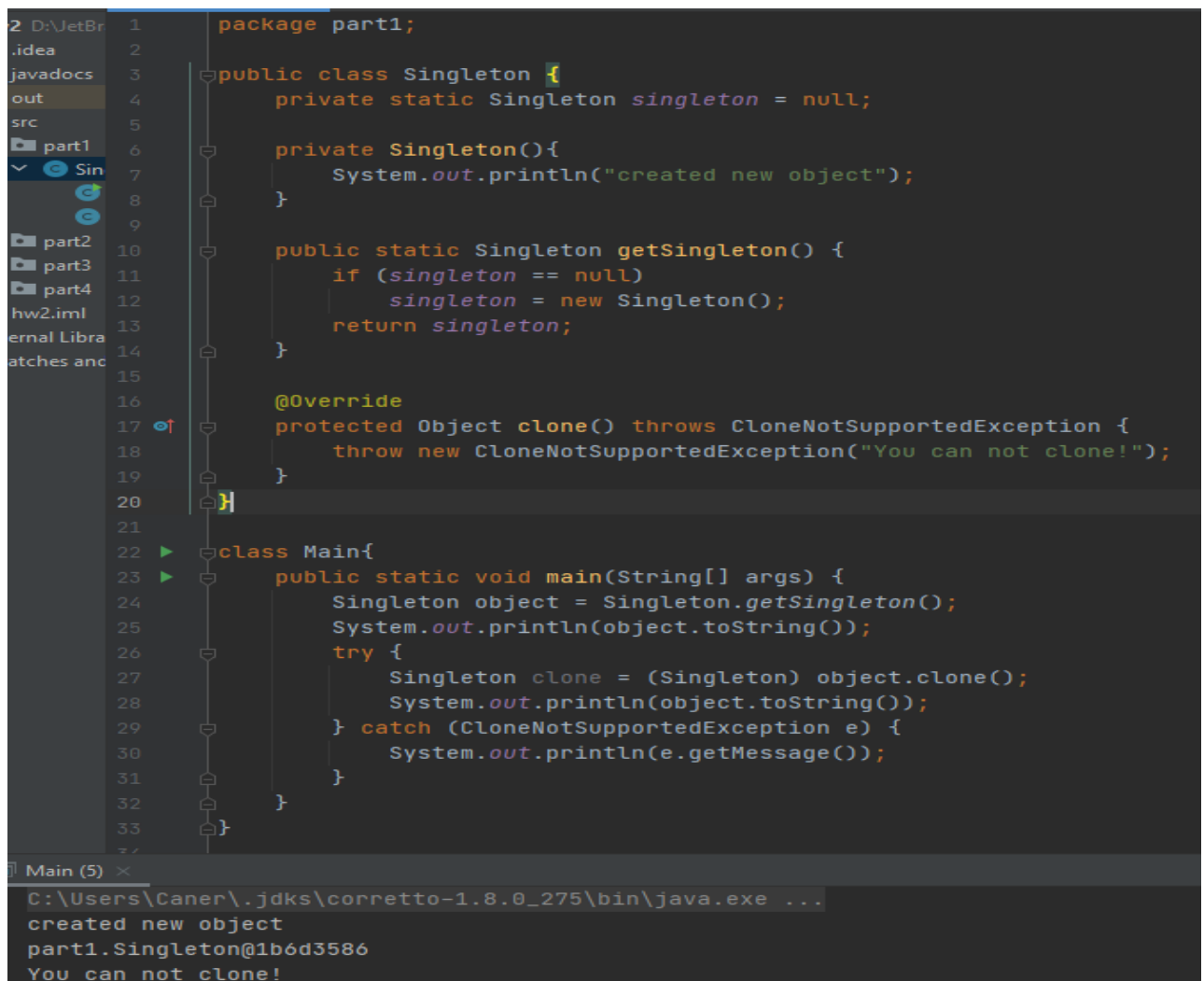
```
1 package part1;
2
3 public class Singleton {
4     private static Singleton singleton = null;
5
6     private Singleton(){
7         System.out.println("created new object");
8     }
9
10    public static Singleton getSingleton() {
11        if (singleton == null) {
12            singleton = new Singleton();
13        }
14        return singleton;
15    }
16
17    @Override
18    protected Object clone() throws CloneNotSupportedException {
19        return super.clone();
20    }
21
22    class Main{
23        public static void main(String[] args) {
24            Singleton object = Singleton.getSingleton();
25            System.out.println(object.toString());
26            try {
27                Singleton clone = (Singleton) object.clone();
28                System.out.println(object.toString());
29            } catch (CloneNotSupportedException e) {
30                System.out.println(e.getMessage());
31            }
32        }
33    }
34 }
```

Output:

```
C:\Users\Caner\.jdk\corretto-1.8.0_275\bin\java.exe ...
created new object
part1.Singleton@1b6d3586
part1.Singleton

Process finished with exit code 0
```

2. While overriding the Clone method, we can prevent this by throwing an exception directly. In addition, if the Singleton class does not implement the Cloneable interface, the clone () method is not called.



```
1 package part1;
2
3 public class Singleton {
4     private static Singleton singleton = null;
5
6     private Singleton(){
7         System.out.println("created new object");
8     }
9
10    public static Singleton getSingleton() {
11        if (singleton == null)
12            singleton = new Singleton();
13        return singleton;
14    }
15
16    @Override
17    protected Object clone() throws CloneNotSupportedException {
18        throw new CloneNotSupportedException("You can not clone!");
19    }
20 }
21
22 class Main{
23     public static void main(String[] args) {
24         Singleton object = Singleton.getSingleton();
25         System.out.println(object.toString());
26         try {
27             Singleton clone = (Singleton) object.clone();
28             System.out.println(object.toString());
29         } catch (CloneNotSupportedException e) {
30             System.out.println(e.getMessage());
31         }
32     }
33 }
```

Output:

```
created new object
part1.Singleton@1b6d3586
You can not clone!
```

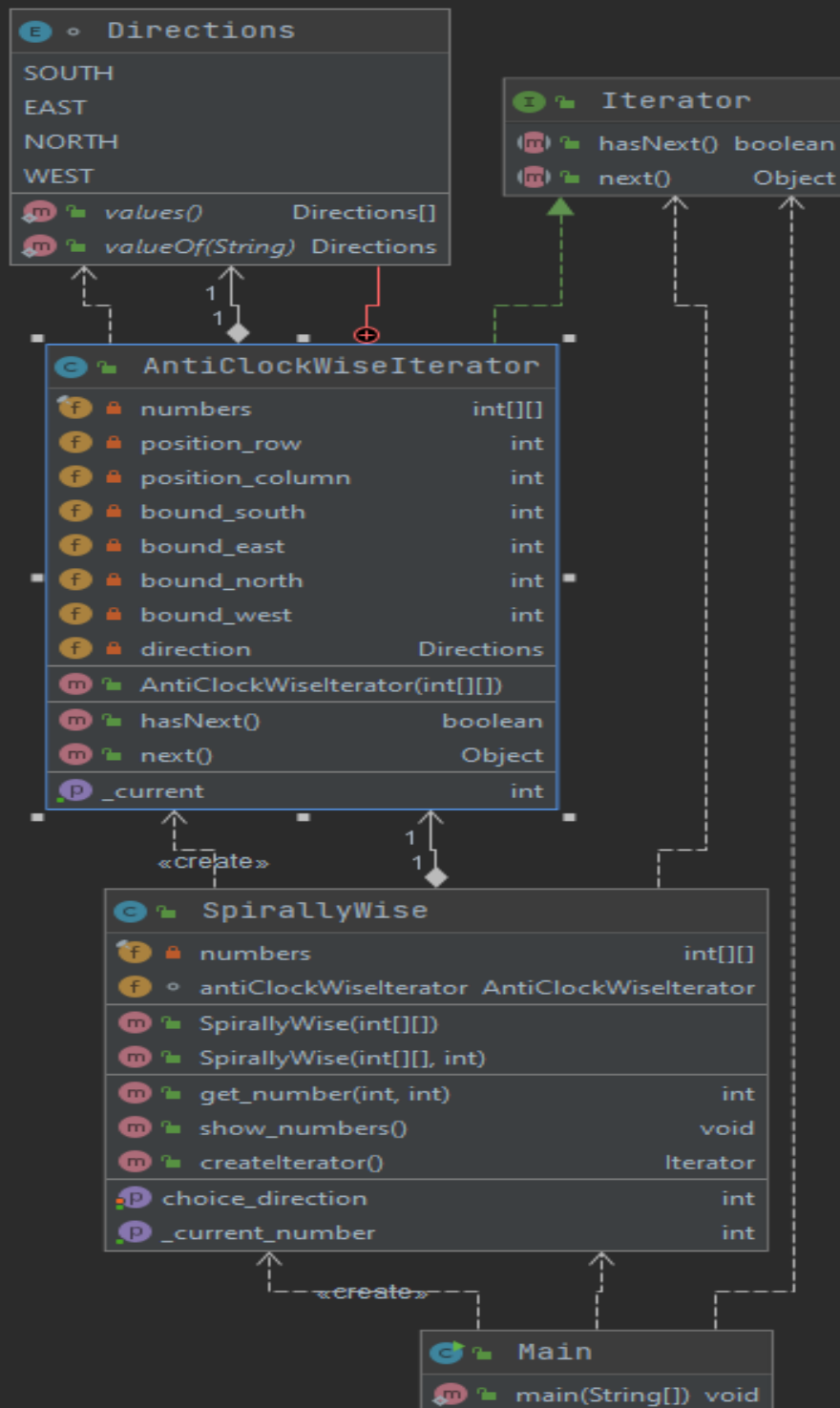
3. There will be no change in the first question. In the second question, the only way is to override the clone () method in subclass and throw an exception. as we did in the example.

## Part 2

The iterator interface has been written for this part. This interface has `hasNext` and `next` methods. There are two classes that can be implemented from this interface and solve the problem we want. These classes should allow us to navigate in two different ways on the given matrix. Only one of these has been implemented. The choice here is that it can move spirally counterclockwise. This class is the `AntiClockWiseIterator` class and is implemented from the `Iterator` interface. If we look at the content of the class, we see that the directions are kept in enum type. because the iterator must have a tendency to go in 4 directions in order to spiral around in the matrix. Constraints are necessary as the matrix will have boundaries in the directions the iterator can travel. four different parameters were kept for these. These parameters hold the indexes of the borders that can be reached in all directions. matrix is kept as a two-dimensional array. The `hasNext` method works by checking the boundaries of its position relative to the direction it will go. `Next` method calls the `hasNext` method to see if it can go according to the direction it will go. If it can't go in the direction it will go, it returns null. If it can go, it does the process of going, and if it reaches the limit, it changes direction and updates its boundaries.

The `SpirallyWise` class was created in order to use this class and other expected clockwise motion iterator and to provide the iterator design. This class will basically have two different iterators (only one of them is implemented). A special object is kept in it that we will use as an iterator. When the `createIterator` method is called, an iterator is created according to the desired type.

```
C:\Users\Caner\.jdk\corretto-1.8.0_275\bin\java.exe ...  
Test For Iterator :  
1 5 9 13 14 15 16 12 8 4 3 2 6 10 11 7  
New Test For Show Numbers New Array2D :  
show all numbers : 1 5 9 13 17 18 19 20 16 12 8 4 3 2 6 10 14 15 11 7  
  
Process finished with exit code 0
```



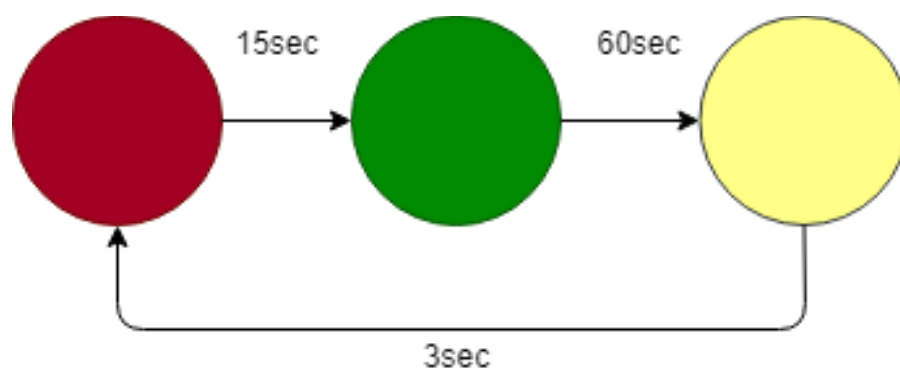
### Part 3

The design to be used for this part is state design pattern.

In state design pattern a class behavior changes based on its state. This type of design pattern comes under behavior pattern. In State pattern, we create objects which represent various states and a context object whose behavior varies as its state object changes.

Here, the state interface and red yellow green methods are defined for three different situations that the traffic light will do. Three classes are defined to implement each state, and these classes are implemented from the state interface. realizes these methods according to their situation. The TrafficLight class was written, where we can use these situations and realize traffic lights. This class holds State objects corresponding to each state. These objects perform the situations expected of us sequentially. In this way, State Design Pattern is provided.

Then, the observer design was used to perform the update from mobes. Subject interface and Observer interface were written for this. Subject interface can update the subscribed traffic light according to the information received from the mobile. Observer interface is for updating. The Hitech technology used to update the traffic light according to the information from Mobes has been implemented with the HiTech class. This class implements the Subject interface and updates the subscribers according to the obsernal design design. TrafficLight class is implemented from Observer interface since it will be the traffic light that is requested to be updated. This way, changes can be updated.



```
C:\Users\Caner\.jdk\corretto-1.8.0_275\bin\java.exe ...
```

```
----test1----
```

```
Traffic light is Red
```

```
Traffic light switches green after 15 seconds
```

```
Traffic light is Green
```

```
Traffic light switches yellow after 60 seconds
```

```
Traffic light is Yellow
```

```
Traffic light switches red after 3 seconds
```

```
Traffic light is Red
```

```
----change----
```

```
Traffic light is Red
```

```
Traffic light switches green after 15 seconds
```

```
Traffic light is Green
```

```
Traffic light switches yellow after 90 seconds
```

```
Traffic light is Yellow
```

```
Traffic light switches red after 3 seconds
```

```
Traffic light is Red
```

```
----change 2----
```

```
Traffic light is Red
```

```
Traffic light switches green after 15 seconds
```

```
Traffic light is Green
```

```
Traffic light switches yellow after 60 seconds
```

```
Traffic light is Yellow
```

```
Traffic light switches red after 3 seconds
```

```
Traffic light is Red
```

```
----wrong test----
```

```
Traffic light is red. You have not switch to yellow
```

```
Traffic light is red. You have not switch to yellow
```

```
Traffic light is Red
```

```
Traffic light switches green after 15 seconds
```

```
Traffic light switches yellow after 60 seconds
```

```
Traffic light is yellow. You have not switch to yellow
```

```
Traffic light is Yellow
```

```
Traffic light is yellow. You have not switch to yellow
```

```
Traffic light is yellow. You have not switch to green
```

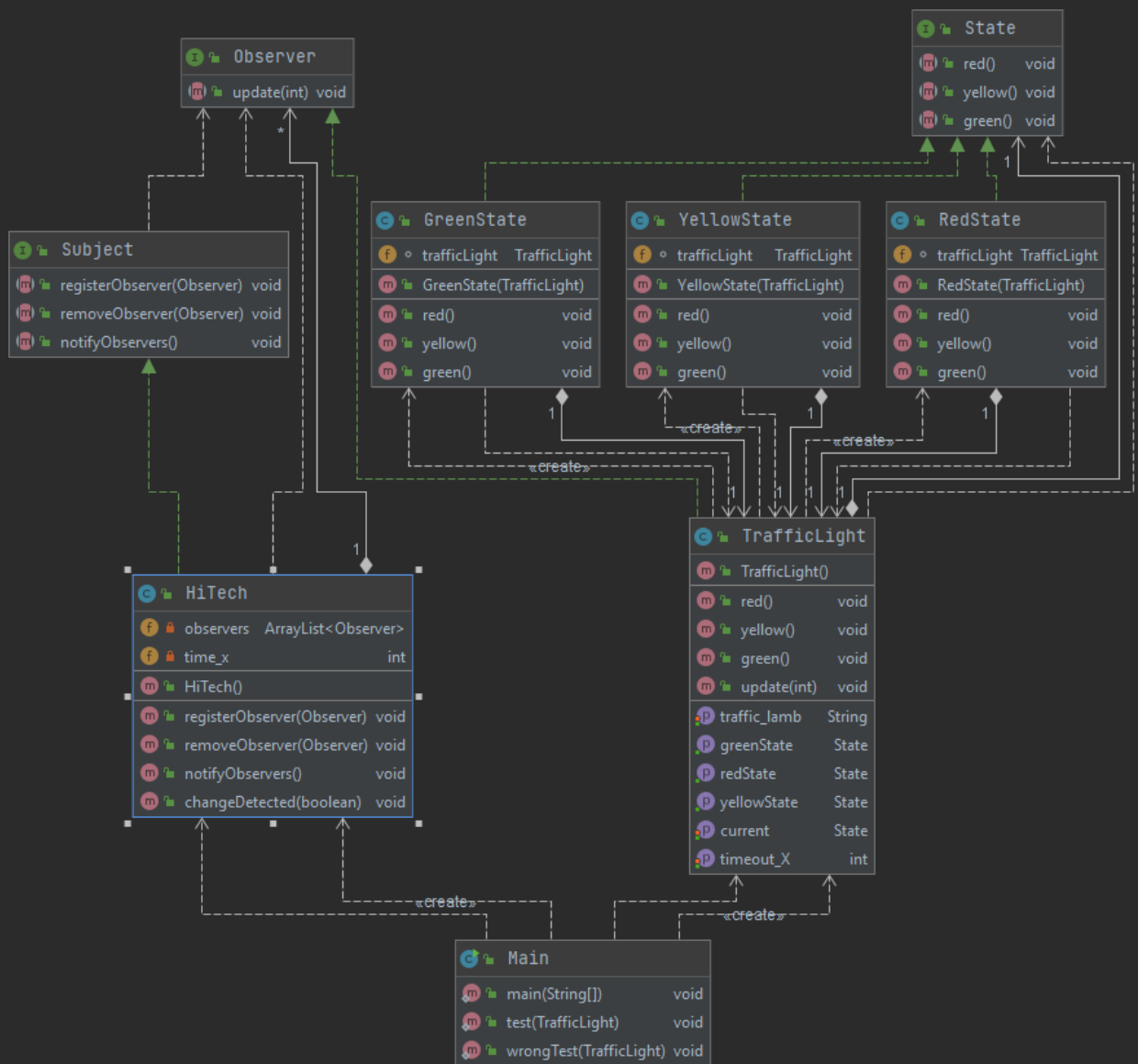
```
Traffic light is yellow. You have not switch to green
```

```
Traffic light is Yellow
```

```
Traffic light switches red after 3 seconds
```

```
Traffic light is Red
```

```
Process finished with exit code 0
```





## Part 4 A

Here, a database management is expected from an interface that we know its methods and have access to. The interface through which we can access its information is the ITable interface. The class whose content we cannot see and change is the DataBaseTable class that is implemented from this interface. We are expected to use this class to learn how to read and write on the matrix and to learn the number of rows and columns. Here we will use proxy synchronous design design. These are the operations that are synchronized but wanted to be done simultaneously over threads.

We will use the interface and class we want to use via Proxy. For this reason, the ProxyClass class was written. It holds an element from the ITable interface. Since we cannot access the DataBaseTable class, we must perform operations through the interface and proxy. That's why we implement it from the InvocationHandler interface. This allows us to use the invoke () method. Thanks to this method, we will be able to perform our transactions. we rewrite this method for ourselves. There are some synchronization problems to be aware of here. Our first problem is that threads that want to write should not conflict with threads that are reading. because a read operation made during the writing process may give wrong results. therefore an object of class Reentrantlock is used. This object will give us the mutex task. Synchronization makes if there is a thread that is reading and a thread is writing, they wait for each other. that is, it prevents the writing operation during reading, and prevents the reading operation during the writing process.

```
C:\Users\Caner\.jdk\corretto-1.8.0_275\bin\java.exe ...
```

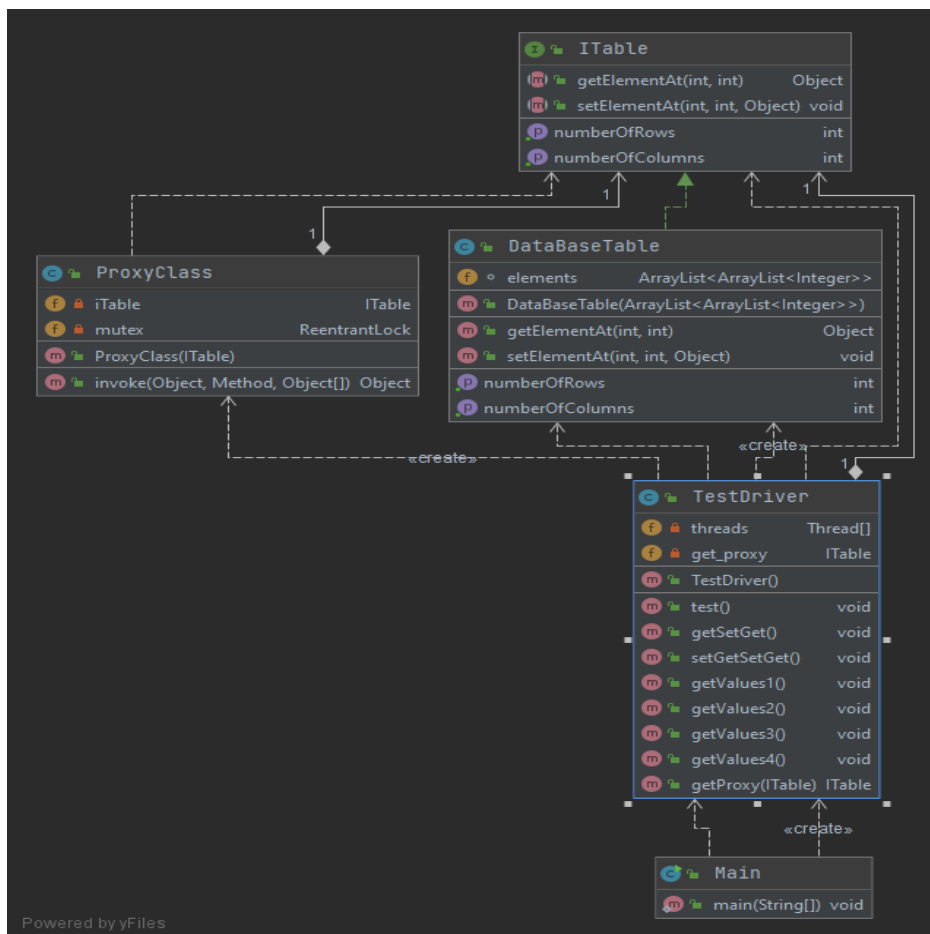
```
getelement girls Thread-0  
Thread-3 getColumns : 3  
Thread-1 getRows : 2  
getelement cikisThread-0  
Thread-1 getColumns : 3  
Thread-7 getColumns : 3  
Thread-0 getelement : 2  
setelement girls Thread-2  
Thread-5 getRows : 2  
Thread-7 getRows : 2  
setelement cikis Thread-2  
getelement girls Thread-2  
getelement cikisThread-2  
Thread-2 ilk degisimi: getelement : -1  
setelement girls Thread-9  
setelement cikis Thread-9  
getelement girls Thread-9  
getelement cikisThread-9  
Thread-9 ilk degisimi: getelement : -1  
getelement girls Thread-4  
getelement cikisThread-4  
Thread-4 getelement : -1  
setelement girls Thread-6  
setelement cikis Thread-6  
getelement girls Thread-6  
getelement cikisThread-6  
Thread-6 ilk degisimi: getelement : -1  
setelement girls Thread-0  
setelement cikis Thread-0  
getelement girls Thread-8  
getelement cikisThread-8  
Thread-8 getelement : -3  
setelement girls Thread-2  
setelement cikis Thread-2  
getelement girls Thread-2  
getelement cikisThread-2  
Thread-2 ikinci degisimi : getelement : -2  
setelement girls Thread-9  
setelement cikis Thread-9  
getelement girls Thread-9
```

```

getelement cikisThread-9
Thread-9 ikinci degisimi : getelement : -2
setelement giris Thread-4
setelement cikis Thread-4
getelement giris Thread-4
getelement cikisThread-4
Thread-4 sonucu : getelement : -3
setelement giris Thread-6
setelement cikis Thread-6
getelement giris Thread-6
getelement cikisThread-6
Thread-6 ikinci degisimi : getelement : -2
getelement giris Thread-0
getelement cikisThread-0
Thread-0 sonucu : getelement : -2
setelement giris Thread-8
setelement cikis Thread-8
getelement giris Thread-8
getelement cikisThread-8
Thread-8 sonucu : getelement : -3

Process finished with exit code 0

```



## Part 4 B

Threads that will do the writing process are given priority in the operations done here. It is a writing priority solution to a classic reading and writing problem. An element that keeps the number of readers can be solved thanks to an element that keeps the number of printers and an element that holds the writing requests. Reading threads wait for those who write and want to do. Those who want to write should wait for the printers. Here, locking and waiting are implemented with wait method and notifyAll methods. ReadUnlock method to open readlock method for read lock, writeUnlock method to open writeLock for write lock.

```
C:\Users\Caner\.jdk\corretto-1.8.0_275\bin\java.exe ...
setelement giris Thread-9
Thread-1 getRows : 2
Thread-1 getColumns : 3
Thread-3 getColumns : 3
setelement cikis Thread-9
setelement giris Thread-6
setelement cikis Thread-6
Thread-7 getColumns : 3
setelement giris Thread-2
Thread-7 getRows : 2
Thread-5 getRows : 2
setelement cikis Thread-2
getelement giris Thread-2
getelement giris Thread-4
getelement cikisThread-2
getelement giris Thread-6
getelement giris Thread-0
getelement giris Thread-9
getelement giris Thread-8
getelement cikisThread-9
getelement cikisThread-0
getelement cikisThread-6
Thread-6 ilk degisimi: getelement : -1
Thread-2 ilk degisimi: getelement : -1
getelement cikisThread-4
Thread-0 getelement : -1
Thread-9 ilk degisimi: getelement : -1
getelement cikisThread-8
Thread-4 getelement : -1
setelement giris Thread-9
Thread-8 getelement : -1
setelement cikis Thread-9
setelement giris Thread-8
setelement cikis Thread-8
setelement giris Thread-0
setelement cikis Thread-0
setelement giris Thread-4
setelement cikis Thread-4
setelement cikis Thread-4
setelement giris Thread-6
setelement cikis Thread-6
setelement giris Thread-2
setelement cikis Thread-2
getelement giris Thread-2
getelement cikisThread-2
getelement giris Thread-8
getelement giris Thread-4
getelement cikisThread-8
Thread-2 ikinci degisimi : getelement : -2
getelement giris Thread-6
getelement giris Thread-0
getelement giris Thread-9
getelement cikisThread-0
getelement cikisThread-6
Thread-8 sonucu : getelement : -2
getelement cikisThread-4
Thread-4 sonucu : getelement : -2
Thread-6 ikinci degisimi : getelement : -2
Thread-0 sonucu : getelement : -2
getelement cikisThread-9
Thread-9 ikinci degisimi : getelement : -2
Process finished with exit code 0
```

