TITLE

Laboratory 4. Modeling and Verifying Concurrency I

PREREQUISITES

- OOP basic knowledge
- Java programming language basic knowledge
- Algorithms

RESOURCES

- Course slides
- Java Tutorial
- Java Concurrency

LABORATORY INSTRUCTIONS

A common event may occur when more than one running threads shares resources. We can see many concurrent applications that read/write data in the same data structure or having access to the same database or files. Therefore, we can have data inconsistency or even errors. In order to solve this type of problem, some mechanisms need to be implemented.

The situation when multiple threads are trying to access the same resource at the same time is called a **race condition**. For example, imagine that you have multiple threads that try to change a shared variable's value, the final value of the variable will depend on the order that the threads are executed and when threads will not have access to the value of the variable that is in the other thread's local caches.

The solution to the problem is based on the **critical section** concept. A critical section is a block where access to a shared resource is can be permitted only to a thread at a time. To be able to implement critical sections, any programming language that supports concurrent programming offers to the programmers some synchronization mechanisms.

In Java, we can find two basic synchronization mechanisms:

- **synchronized** keyword
- The **Lock** interface

The **synchronized** keyword can be used in four types of blocks:

1. Instance methods

```
public class MyCounter {
  private int count = 0;

  public synchronized void add(int value){

      this.count += value;

  }

}
```

!!! The synchronized method will be executed by one running thread at a time per instance.

2. Static methods

```
public static MyStaticCounter{

  private static int count = 0;

  public static synchronized void add(int value){

      count += value;

  }

}
```

!!! Only one running thread can execute the method for the same class.

3. Code blocks inside the instance methods

```
public void add(int value){

    synchronized(this){

       this.count += value;

    }

  }
```

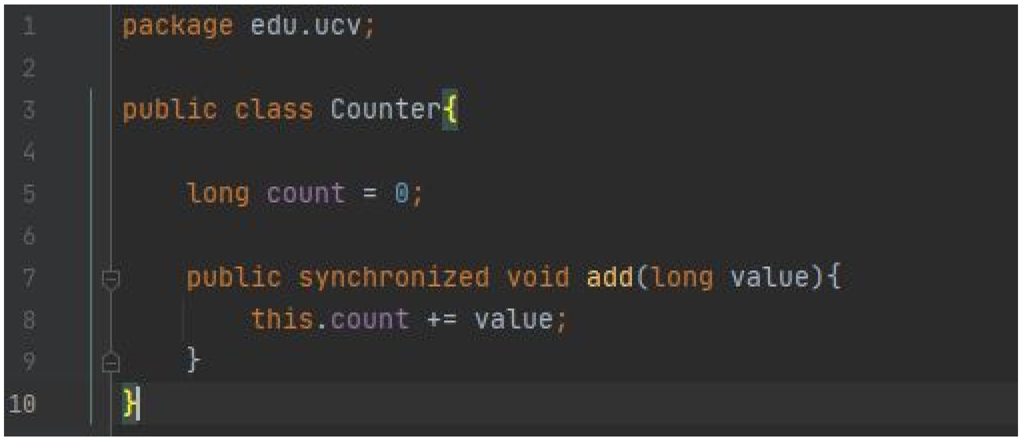
4. Code blocks inside static methods

```
 public class MyClass {

    public static void log(String msg1, String msg2){

       synchronized(MyClass.class){

          log.writeln(msg1);

          log.writeln(msg2);

       }

    }
```

}

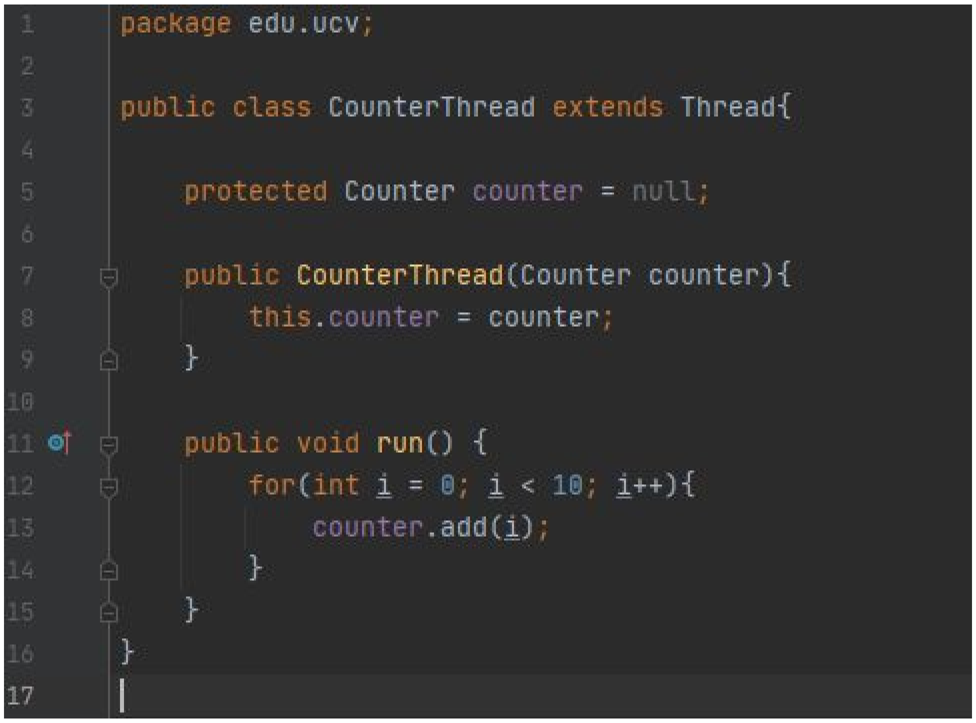
Counter Example

Counter.java



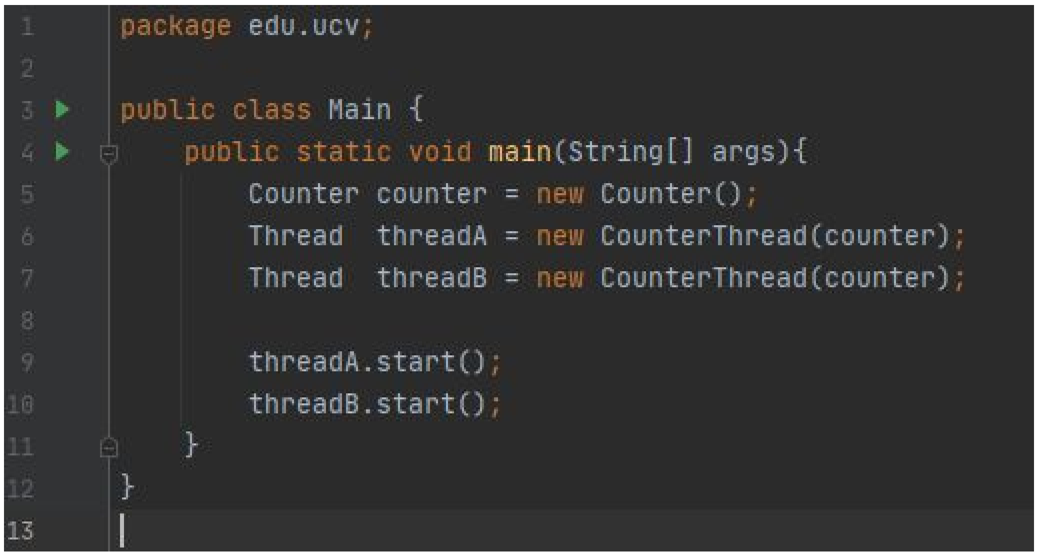
```java
package edu.ucv;

public class Counter{

    long count = 0;

    public synchronized void add(long value){
        this.count += value;
    }
}
```

CounterThread.java



```java
package edu.ucv;

public class CounterThread extends Thread{

    protected Counter counter = null;

    public CounterThread(Counter counter){
        this.counter = counter;
    }

    public void run() {
        for(int i = 0; i < 10; i++){
            counter.add(i);
        }
    }
}
```

Main.java



```java
package edu.ucv;

public class Main {
    public static void main(String[] args){
        Counter counter = new Counter();
        Thread  threadA = new CounterThread(counter);
        Thread  threadB = new CounterThread(counter);

        threadA.start();
        threadB.start();
    }
}
```

LABORATORY TASKS

1. Implement the Counter example. Add proper logging messages to have a better view of how the common resource is accessed.
2. For the example above, create two different instances of the Counter class. What do you observe?