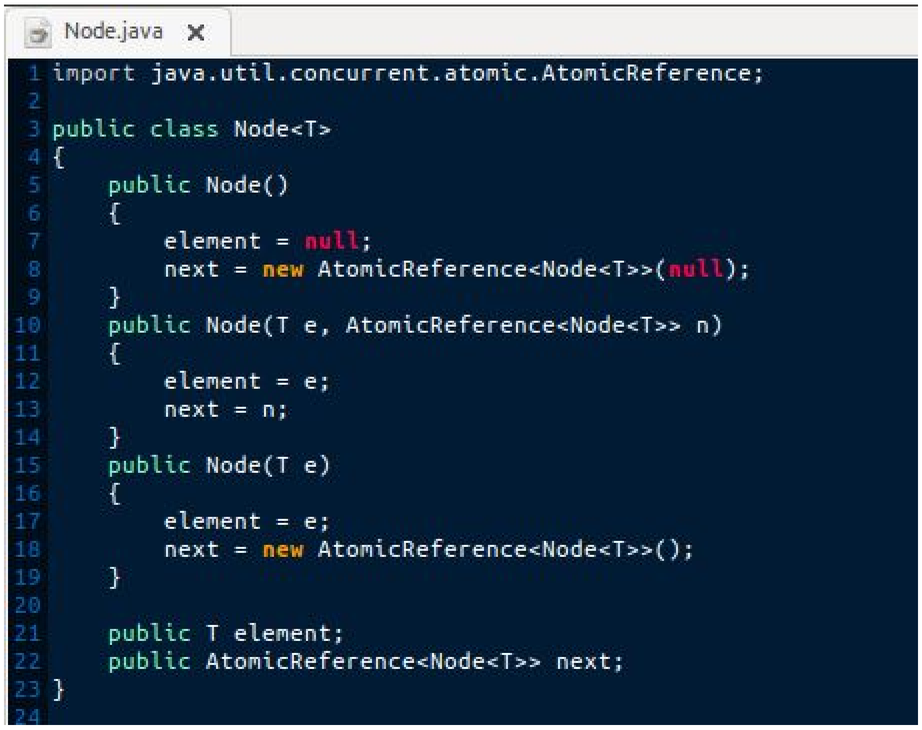
COP6616 Homework Assignment 1 Report

In this assignment, we are asked to implement a lock-free stack using atomic and volatile functionality of the programming languages. I used Java8 and its AtomicReference class to implement the stack. The AtomicReference class also includes the functionality of volatile keyword.

In this report, implementation details in every class are explained one-by-one.

Node.java:



Node.java

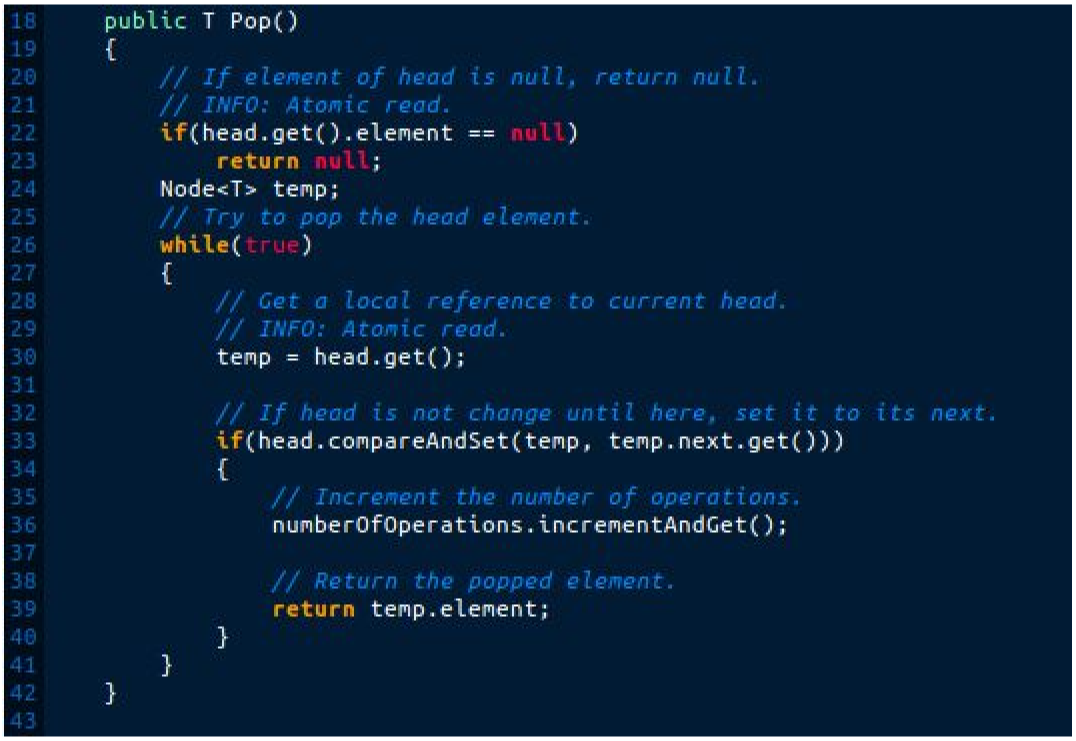
```java
import java.util.concurrent.atomic.AtomicReference;

public class Node<T>
{
    public Node()
    {
        element = null;
        next = new AtomicReference<Node<T>>(null);
    }
    public Node(T e, AtomicReference<Node<T>> n)
    {
        element = e;
        next = n;
    }
    public Node(T e)
    {
        element = e;
        next = new AtomicReference<Node<T>>();
    }

    public T element;
    public AtomicReference<Node<T>> next;
}
```

In this class, nodes of the stack is implemented. Every node has an AtomicReference to the next node. AtomicReference is a class that performs atomic operations(get, set, compareAndSet etc.) on a given reference. Since threads are not interested in changing the “element” field, they are not causing a race condition. Therefore, the elements of the nodes are not atomic.

Stack.java:

In this part, two main methods of *Stack* class will be reviewed.

`Pop():`

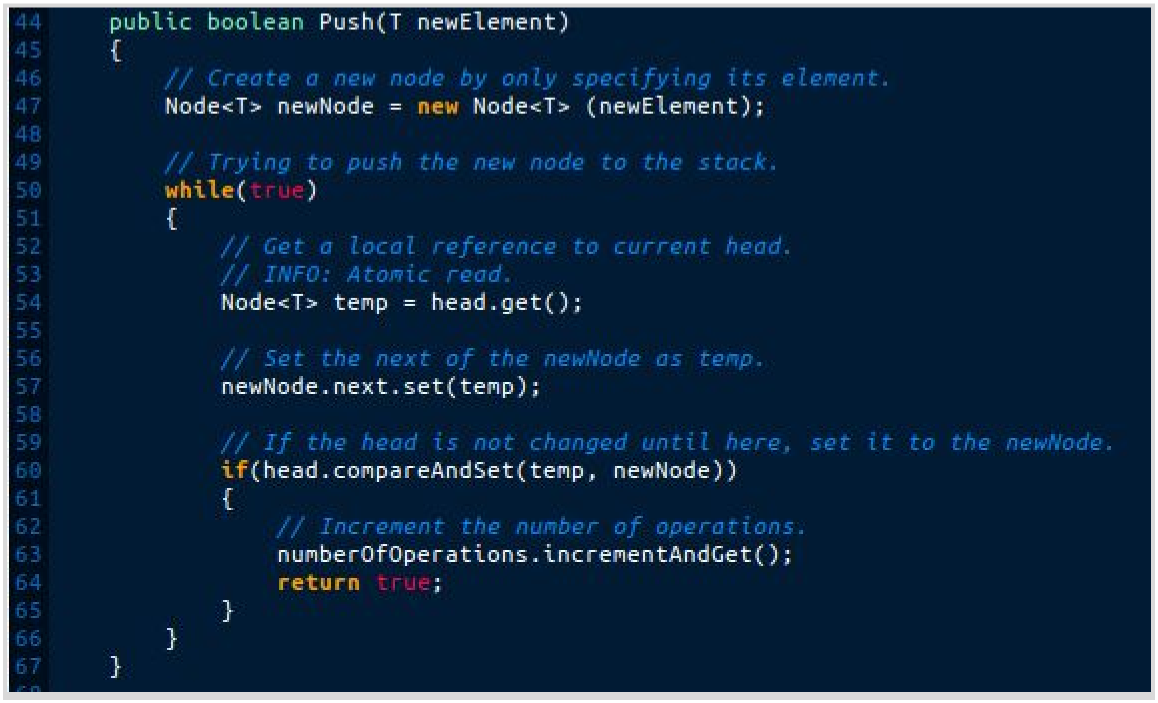


```java
    public T Pop()
    {
        // If element of head is null, return null.
        // INFO: Atomic read.
        if(head.get().element == null)
            return null;
        Node<T> temp;
        // Try to pop the head element.
        while(true)
        {
            // Get a local reference to current head.
            // INFO: Atomic read.
            temp = head.get();

            // If head is not change until here, set it to its next.
            if(head.compareAndSet(temp, temp.next.get()))
            {
                // Increment the number of operations.
                numberOfOperations.incrementAndGet();

                // Return the popped element.
                return temp.element;
            }
        }
    }
```

In this function, we need to change the shared reference `head` to its `next` without using locks. To accomplish this, the features of AtomicReference class will be used. Bear in mind that the `get` and `compareAndSet` functions are atomic functions.

Firstly, the `Pop()` method checks the stack is empty or not. If the stack is empty, it returns `null`. Otherwise, it goes into a infinite loop that will terminate once it accomplishes to set the `head` properly. It takes a local copy of the `head` and in the if clause it compares the previously fetched local copy with the current head atomically. If the current head and the local copy is the same, it sets the head to its `next`. After setting the head, it atomically increments the `numberOfOperations` by one and returns the popped element.

`Push(T newElement):`

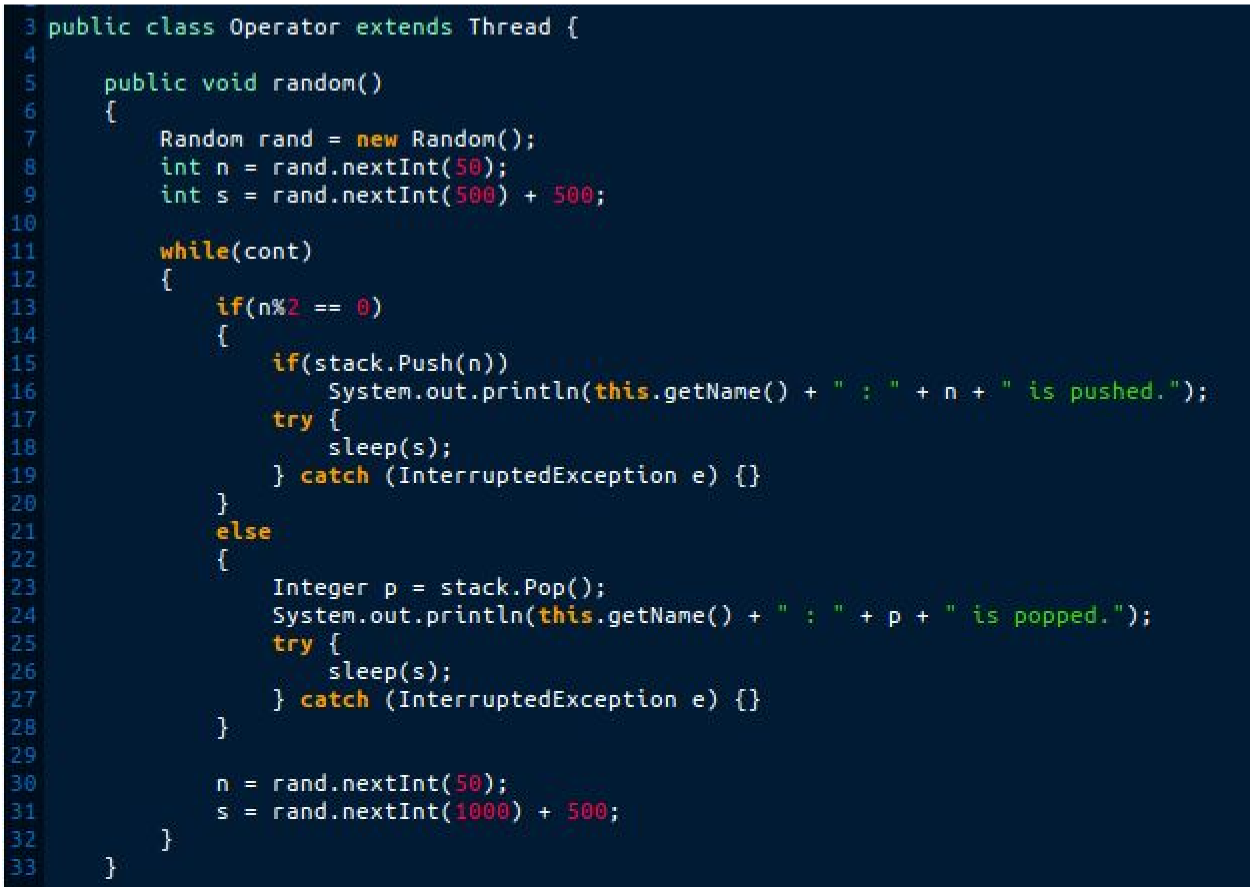


```java
    public boolean Push(T newElement)
    {
        // Create a new node by only specifying its element.
        Node<T> newNode = new Node<T> (newElement);

        // Trying to push the new node to the stack.
        while(true)
        {
            // Get a local reference to current head.
            // INFO: Atomic read.
            Node<T> temp = head.get();

            // Set the next of the newNode as temp.
            newNode.next.set(temp);

            // If the head is not changed until here, set it to the newNode.
            if(head.compareAndSet(temp, newNode))
            {
                // Increment the number of operations.
                numberOfOperations.incrementAndGet();
                return true;
            }
        }
    }
```

In this functions, we need to change the shared `head` without using locks.

Firstly, a new node is created by setting its `element` field. After that, a local copy of the `head` is created and the `next` field of the `newNode` is updated. Then, by the help of `compareAndSet` method, the `head` is changed atomically. As a final step, the `numberOfOperation` is incremented atomically and the method returns `true`.

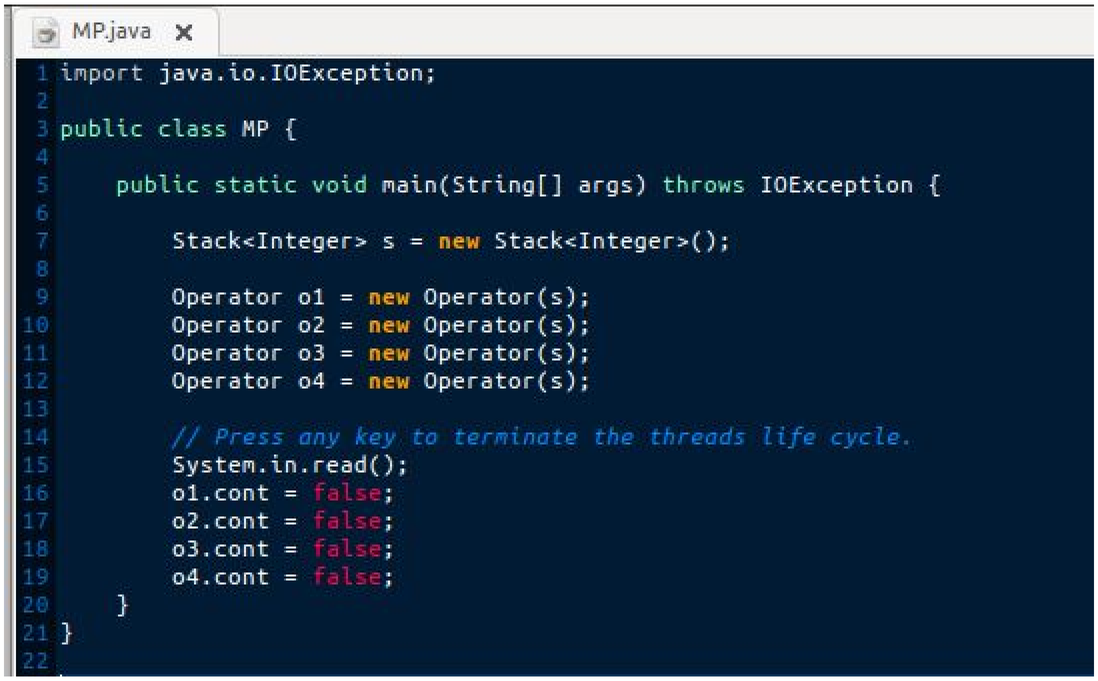
Operator.java:



```java
public class Operator extends Thread {

    public void random()
    {
        Random rand = new Random();
        int n = rand.nextInt(50);
        int s = rand.nextInt(500) + 500;

        while(cont)
        {
            if(n%2 == 0)
            {
                if(stack.Push(n))
                    System.out.println(this.getName() + " : " + n + " is pushed.");
                try {
                    sleep(s);
                } catch (InterruptedException e) {}
            }
            else
            {
                Integer p = stack.Pop();
                System.out.println(this.getName() + " : " + p + " is popped.");
                try {
                    sleep(s);
                } catch (InterruptedException e) {}
            }

            n = rand.nextInt(50);
            s = rand.nextInt(1000) + 500;
        }
    }
```

This class inherits the `thread` class of Java. Every instance of this class will be a thread and the thread starts to run when it’s created. The main method of this class, which is `run()` calls the `random()` method. In this method, the thread creates two random number to determine whether it will pop or push and to determine how long it will sleep.

MP.java:



```java
import java.io.IOException;

public class MP {

    public static void main(String[] args) throws IOException {

        Stack<Integer> s = new Stack<Integer>();

        Operator o1 = new Operator(s);
        Operator o2 = new Operator(s);
        Operator o3 = new Operator(s);
        Operator o4 = new Operator(s);

        // Press any key to terminate the threads life cycle.
        System.in.read();
        o1.cont = false;
        o2.cont = false;
        o3.cont = false;
        o4.cont = false;
    }
}
```

The main method is in this class. In the main method, threads are created and invoked. After that, the main method waits for a user input to terminate the process.

Comments on Linearization

The concerned methods that affects the linearity in lock-free stack implementation is Pop and Push methods. To prove that this implementation is linearizable, we have to show that the linearization points of these methods will be in order.

Linearization point for `Pop()` function :

`head.compareAndSet(temp, temp.next.get())`

Linearization point for `Push()` function :

`head.compareAndSet(temp, newNode)`

No thread will be able to enter the linearization point of the methods at the same time, since `compareAndSet()` method is an atomic method. This fact shows that push and pop operations won't interfere with each other that may cause an unwanted side effect.