# Lab Session 11
# Separation Logic and Concurrency

Software Verification

Nova School of Science and Technology
Mário Pereira `mjp.pereira@fct.unl.pt`

Version of November 17, 2025

For verification tasks that involve the concurrency control mechanisms you will have to install the spec `java.util.concurrent.locks.javaspec` (available online in CLIP) for the package `java.util.concurrent.locks` by copying it to the `bin/rt/` folder of your local verifast installation and append the `rt.jarspec` file with the name of the spec file.

## 1 Exercises

**Exercise 1.** Consider the `Stack` implementation from Appendix A. Implement a concurrent version of `Stack` using monitors, such that the `pop` method will *wait* for the stack to be non-empty. □

**Exercise 2.** Consider the `Queue` implementation from Appendix B. Implement a concurrent version of `Queue` using monitors, such that the `take` method will *wait* for the queue to be non-empty. □

**Exercise 3.** Implement a concurrent version of the `Queue` using monitors, such that the `take` method will *wait* for the queue to be non-empty and the `add` method will *wait* for the queue to be not full. □

**Exercise 4.** Implement (in Java) and specify using Verifast the class `PosQueue` which represents a queue of positive numbers. The API of the queue is:

```
//Queue based on a circular buffer.
class Queue {

  //creates a new Queue with capacity max.
  Queue(int max);

  //places the int v at the end of this Queue
  void enqueue(int v);

  //retrieves the element at the start of this Queue.
  int dequeue();

  //returns true if this Queue has reached its capacity.
  boolean isFull();

  //returns true if this Queue has no elements.
  boolean isEmpty();

}
```

You will likely want to define a predicate of the form: `predicate QueueInv(Queue q; int n, int m)`, where `n` is the number of elements in the queue and `m` is the queue capacity. Note that your invariant will have to characterize the array using `array_slice` and `array_slice_deep` predicates in terms of the head index, the tail index and the beginning (index 0) and the length of the array. □

# A Stack Implementation

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

//@ predicate Node(Node n; Node nxt, int v) = n.next |-> nxt &*& n.val |-> v;

public class Node {
    Node next;
    int val;

    public Node()
    //@ requires true;
    //@ ensures Node(this, null, 0);
    {
        next = null;
        val = 0;
    }

    public void setnext(Node n)
    //@ requires Node(this, _, ?vv);
    //@ ensures Node(this, n, vv);
    {
        next = n;
    }
    public void setval(int v)
    //@ requires Node(this, ?nn, _);
    //@ ensures Node(this, nn, v);
    {
        val = v;
    }

    public Node getnext()
    //@ requires Node(this, ?nn, ?vv);
    //@ ensures Node(this, nn, vv) &*& result == nn;
    {
        return next;
    }

    public int getval()
    //@ requires Node(this, ?nn, ?vv);
    //@ ensures Node(this, nn, vv) &*& result == vv;
    {
        return val;
    }
}

/*@
  predicate List(Node n; list<int> elems) =
    n == null ? (emp &*& elems == nil) :
```

```
                Node(n,?nn,?v) &*& List(nn,?tail) &*& elems == cons(v,tail);

  predicate StackInv(Stack s; list<int> l, int max) =
    s.head |-> ?h &*& List(h, l) &*& s.max |-> max; @*/

public class Stack {
    Node head;
    int max;

    public Stack(int max)
    //@ requires max > 0;
    //@ ensures StackInv(this, nil, max);
    {
        head = null;
        this.max = max;
    }

    public void push(int v)
    //@ requires StackInv(this, ?l, ?m);
    //@ ensures StackInv(this, cons(v, l), m);
    {
        Node n = new Node();
        n.setval(v);
        n.setnext(head);
        head = n;
    }

    public int pop()
    //@ requires StackInv(this, cons(?x, ?vs), ?m);
    //@ ensures StackInv(this, vs, m) &*& result == x;
    {
        //@ open StackInv(this, _, _);
        //@ open List(head, _);
        int v = head.getval();
        head = head.getnext();
        return v;
    }

    public boolean isEmpty()
    //@ requires StackInv(this, ?l, ?m);
    //@ ensures StackInv(this, l, m) &*& result == (l == nil);
    {
        //@ open StackInv(this, _, _);
        //@ open List(head, _);
        return head == null;
    }

    public void clear ()
    //@ requires StackInv(this, ?l, ?m);
```

```
    //@ ensures StackInv(this, nil, m);
    {
        head = null;
    }
}
```

# B Queue Implementation

```
/*@

  predicate Cell (Cell c; Cell nxt, int v) =
    c.next |-> nxt &*& c.content |-> v;

  predicate lseg (Cell from, Cell to; list <int> l) =
    from == to ?
      l == nil :
      from != null &*& from.content |-> ?v &*&
      from.next |-> ?next &*& lseg(next, to, ?nvs) &*&
      l == cons (v, nvs);

  predicate QueueInv (Queue q; list<int> vs) =
    q.first |-> ?first &*& q.last |-> ?last &*& q.length |-> ?ll &*&
    last == null ?
      first == null &*& vs == nil &*& ll == 0:
      lseg(last, null, cons(?v, nil)) &*& lseg(first, last, ?vs1) &*&
      vs == append(vs1, cons(v, nil)) &*& ll == length(vs);

  lemma void cell_null_lseg (Cell c)
    requires Cell(c, null, ?v); //??? &*& c != null
    ensures lseg(c, null, cons(v, nil));
  {
    open Cell(c, null, v);
  }

  lemma void lseg_merge (Cell x, Cell y, Cell w)
    requires lseg(x, y, ?vs1) &*& lseg(y, w, ?vs2) &*& lseg(w, null, ?vs3);
    ensures lseg(x, w, append(vs1, vs2)) &*& lseg(w, null, vs3);
  {
    open lseg(w, null, vs3);
    open lseg(x, y, vs1);
    if (x != y) {
      lseg_merge(x.next, y, w);
    }
    else { }
  }

  lemma_auto void length_one (list<int> vs, int x)
    requires true;
    ensures length(vs) + 1 == length(append(vs, cons(x, nil)));
  {
    length_append(vs, cons(x, nil));
  }

@*/
```

```
class Cell
{
    Cell next;
    int content;

    public Cell (int v, Cell next)
    //@ requires true;
    //@ ensures Cell(this, next, v);
    {
        this.next = next;
        content = v;
    }

    public Cell clone()
    //@ requires Cell(this, ?n, ?v);
    //@ ensures Cell(this, n, v) &*& Cell(result, n, v);
    {
        Cell c = new Cell(this.content, this.next);
        return c;
    }
}

public class Queue
{
    Cell first;
    Cell last;
    int length;

    public Queue ()
    //@ requires true;
    //@ ensures QueueInv(this, nil);
    {
        first = null;
        last = null;
        length = 0;

        //@ close QueueInv(this, nil);
    }

    public void clear ()
    //@ requires QueueInv(this, ?l);
    //@ ensures QueueInv(this, nil);
    {
        //@ open QueueInv(this, l);
        first = null;
        last = null;
        length = 0;
        //@close QueueInv(this, nil);
    }
```

```
public void add (int x)
//@ requires QueueInv(this, ?l);
//@ ensures QueueInv(this, append(l, cons(x, nil)));
{
    Cell cell = new Cell(x, null);

    //@ open QueueInv(this, l);

    if (last == null) {
        last = cell;
        first = cell;
        length = 1;

        //@ cell_null_lseg(cell);
        //@ close QueueInv(this, cons(x, nil));
    }
    else {
        if (last == first) {
            //@ open lseg(first, last, ?vs1);

            //@ Cell old_last = last;
            last.next = cell;
            last = cell;
            //@ open Cell(last, null, x);

            length = 2;

            //@ close lseg(cell, null, cons(x, nil));
            //@ close QueueInv(this, append(l, cons(x, nil)));
        }
        else {
            //@ Cell old_last = last;
            last.next = cell;
            last = cell;

            //@ close Cell(last, null, x);

            length = length + 1;

            //@ cell_null_lseg(last);
            //@ lseg_merge(first, old_last, last);
            //@ close QueueInv(this, _);
        }
    }
}

public boolean isEmpty ()
//@ requires QueueInv(this, ?l);
```

```
    //@ ensures QueueInv(this, l) &*& result == (l == nil);
    {
        return length == 0;
    }

    public int take ()
    //@ requires QueueInv(this, cons(?v, ?vs));
    //@ ensures QueueInv(this, vs) &*& result == v;
    {
        //@ open QueueInv(this, cons(v, vs));
        //@ open lseg(first, last, ?vv);

        if (first == last) {
            int c = first.content;
            this.clear();

            return c;
        }
        else {
            int c = first.content;
            length = length - 1;
            first = first.next;

            return c;
        }
    }
}
```