1999

p4q3
PR
p11q3

# Further Java

## *Question 2*

Describe the facilities in Java for restricting concurrent access to critical regions. Explain how shared data can be protected through the use of objects. [8 marks]

The built-in facilities for restricting concurrency in Java only allow one thread at a time to be executing within the critical region. A different approach is to distinguish *shared* and *exclusive* access to a critical region; any number of *readers* may share access at the same time, but only one *writer* may acquire exclusive access (excluding any readers while it does so).

Specify a MultiSync class in Java with methods to acquire and release both read and write access, and sketch its implementation. [6 marks]

Derive a MultiBuffer class that extends MultiSync with methods to store and read a data field, ensuring that any locks are released when a waiting thread is interrupted. Your example may use a simple data field such as an integer but you should explain why such an elaborate scheme is unnecessary in this case. [6 marks]

## Answer

Monitor as list of Threads blocked either because of mutual exclusion or because they have waited explicitly; synchronized statements and methods; wait, notify and notifyAll; while (!ready) wait(); idiom; critical data in fields of a class accessible only through synchronized methods.

See attached code. Credit for correct use of synchronized methods and idiom.

See attached code. Credit for use of try ... finally. No protection is necessary for a value that would be written or read atomically.

```
/*  Readers and writers example
 *  Peter Robinson
 *  February 1999
 */


public class MultiSyncExample {

    public static void main(String args[]) {
      MultiBuffer mb = new MultiBuffer ();
      Thread p = new Producer (mb, 2);
      Thread c = new Consumer (mb, 3);
      c.start ();
      p.start ();
    }
}


class MultiSync {
    int readers = 0;
    int writers = 0;

    synchronized void acquireRead () throws InterruptedException {
      while (writers > 0) wait ();
      readers++;
    }

    synchronized void releaseRead () throws InterruptedException {
      readers--;
      notify ();
    }

    synchronized void acquireWrite () throws InterruptedException {
      while ((writers + readers) > 0) wait ();
      writers++;
    }

    synchronized void releaseWrite () throws InterruptedException {
      writers--;
      notifyAll ();
    }
}


class MultiBuffer extends MultiSync {
    int value;
    boolean valid = false;

    void put (int i) throws InterruptedException {
      try {
          acquireWrite ();
          value = i;
          valid = true;
      }
      finally {
          releaseWrite ();
      };
    }

    int get () throws InterruptedException, IllegalStateException {
      try {
          acquireRead ();
          if (!valid) throw new IllegalStateException ("No data in
buffer");
```

```
            return value;
        }
        finally {
            releaseRead ();
        };
    }
}


class Producer extends Thread {
    MultiBuffer mb;
    int delay;

    Producer (MultiBuffer m, int d) {
      mb = m;
      delay = d * 1000;
    }

    public void run () {
      for (int i = 1; ; i++) {
          try {
            mb.put (i);
            sleep (delay);
          }
          catch (InterruptedException e) {
            System.out.println ("Producer interrupted: " + e);
          };
      };
    }
}


class Consumer extends Thread {
    MultiBuffer mb;
    int delay;

    Consumer (MultiBuffer m, int d) {
      mb = m;
      delay = d * 1000;
    }

    public void run () {
      for (;;) {
          try {
            System.out.println ("Found: " + mb.get ());
            sleep (delay);
          }
          catch (InterruptedException e) {
            System.out.println ("Consumer interrupted: " + e);
          }
          catch (IllegalStateException e) {
            System.out.println ("Consumer failed: " + e);
          };
      };
    }
}
```