064202

THE UNIVERSITY of York

Degree Examinations 2003

DEPARTMENT OF COMPUTER SCIENCE

Concurrent and Real-Time Programming in Java

Time allowed: **One and one half (1.5) hours**

Candidates should answer not more than **two** questions.

An appendix is provided which lists the relevant Java and RTSJ classes needed for this examination.

1 (25 marks)

(i) [5 marks] Explain Bloch's thread safety levels for Java classes.

(ii) [10 marks] Consider the following class:

```
public class RunConcurrent
{
  public RunConcurrent(int maximumNumber)
  {
     // you decide
  }

  public void concurrentExecution(Runnable r)
  {
     // entry protocol -- you decide
     r.run();
     // exit protocol -- you decide
  }

  // any internal methods or fields you need
}
```

Each instance of this class will allow `maximumNumber` threads to call the `concurrentExecution` method. The required synchronization is:

- no call to the `r.run` method is made until `maximumNumber` threads have called `concurrentExecution`, all outstanding calls to the `r.run` method are then made;
- calls to `r.run` should run concurrently not sequentially;
- no return from `concurrentExecution` must occur until all the `r.run` methods have returned.

Show how the `RunConcurrent` class can be implemented. You may assume that only `maximumNumber` threads will call instances of the class and that each thread only makes one call to `concurrentExecution`. You may also ignore the throwing of any `InterruptedException`.

(iii) [10 marks] Show how you would modify the class to cope with the situation where more than `maximumNumber` threads can call instances of the class. The required semantics are that the class only allows `maximumNumber` calls to `concurrentExecution` to call their associated run methods at once. If more than `maximumNumber` threads call `concurrentExecution`, the first `maximumNumber` calls should be allowed to proceed and complete, the other threads should be queued and serviced later when a further `maximumNumber` calls have been made.

2 (25 marks)

(i) [13 marks] Discuss the extent to which the RTSJ asynchronous transfer of control facility is integrated with the Java exception handling facility.

(ii) [12 marks] Consider the following class:

```
import javax.realtime.*;
public class ATCExample
{
  public void methodA() throws AsynchronouslyInterruptedException
  {
    try {
      // complex time consuming algorithm
    }
    catch(InterruptedException e)
    { System.out.println("caught in methodA"); }
    finally { System.out.println("leaving methodA"); }
  }

  public void methodB()
  {
    try {
       methodA();
    }
    catch(InterruptedException e) {System.out.println("caught in methodB");}
    finally { System.out.println("leaving methodB"); }
  }
```

```
  public void methodC() throws AsynchronouslyInterruptedException
  {
    try {
       methodB();
    }
    catch(Exception e) { System.out.println("caught in methodC"); }
    finally { System.out.println("leaving methodC");}
  }

  public void methodD() throws AsynchronouslyInterruptedException
  {
    try {
       methodC();
    }
    catch(Exception e) { System.out.println("caught in methodD");}
    finally { System.out.println("leaving methodD");}
  }


  public void methodE()
  {
    try {
      methodD();
    }
    catch(AsynchronouslyInterruptedException aie)
    { System.out.println("caught in methodE"); }
    finally {System.out.println("leaving methodE");
    }
  }

  public void methodF()
  {
    try {
      methodE();
      System.out.println("methodF finishing");
    }
    catch(Exception aie) {System.out.println("caught in methodF");}
  }
}
```

Suppose that a real-thread `T` calls `methodF` in an instance of this class and that whilst the real-time thread is executing in `methodA` it is interrupted by a call to the `T.interrupt` method. What output does the real-time thread produce? Explain your answer.

3 (25 marks)

(i) [8 marks] Summarize the memory assignment rules for the RTSJ.

(ii) [5 marks] What is the single parent rule and how does it relate to the memory assignment rules?

(iii) [6 marks] Consider the following class:

```
import javax.realtime.*;
public class MemoryAreaExample
{
  public MemoryAreaExample()
  {
    memA = new LTMemory(1024, 1024);
    memB = new LTMemory(1024, 1024);
    memC = new LTMemory(1024, 1024);
    memD = new LTMemory(1024, 1024);
  }

  public void nested()
  {
   memA.enter(r1);
  }

  private Runnable r1 = new Runnable()
    {
      public void run()
      {
        memB.enter(r2);
      }
    };

  private Runnable r2 = new Runnable()
    {
      public void run()
      {
        memC.enter(r3);
      }
    };
```

```
    private Runnable r3 = new Runnable()
      {
        public void run()
        {
          try {
            memA.executeInArea(r4);
          } catch(javax.realtime.InaccessibleAreaException e) ;
        }
      };

    private Runnable r4 = new Runnable()
      {
        public void run()
        {
          memD.enter(r5);
        }
      };

    private Runnable r5 = new Runnable()
      {
        public void run()
        {
          // arbitrary code
        }
      };

    private LTMemory memA;
    private LTMemory memB;
    private LTMemory memC;
    private LTMemory memD;
}
```

A real-time thread calls the `nested` method in an instance of this class whilst it is active in the heap memory area. Show how the scope stack grows and shrinks as a result of executing this method.

(iv) [6 marks] What would happen in the above code if `r3` was replaced with the following

```
    private Runnable r3 = new Runnable()
      {
        public void run()
        {
          memA.enter(r4);
        }
      };
```

064202

THE UNIVERSITY of York

DEPARTMENT OF COMPUTER SCIENCE

Degree Examinations 2003

Concurrent and Real-Time Programming in Java

Model Answers

Model Answers

Question 1 (25 marks)

Part (i) [5 marks]

Book work

Immutable: Instances of the class are constant and cannot be changed. There are, therefore, no thread safety issues.

Thread-safe: Instances of the class are mutable but they can be used safely in a concurrent environment as the methods. All methods provided by the class are properly synchronized either at the interface level or internally within the method.

Conditionally thread-safe: Instances of the class either have methods which are thread-safe, or have methods which are called in sequence with the lock held by the caller.

Thread compatible: Instances of the class provide no synchronization. However, instances of the class can be safely used in a concurrent environment, if the caller provides the synchronization by surrounding each method (or sequence of method calls) with the appropriate lock.

Thread-hostile: Instances of the class should not be used in a concurrent environment even if the caller provides external synchronization. Ideally, classes should not be written which are thread hostile. Typically a thread hostile class is accessing static data or the external environment.

Part (ii) [10 marks]

Unseen problem

```
public class RunConcurrent
{
  public RunConcurrent(int maximumNumber)
  {
     active = 0;
     finished = 0;
     needed = maximumNumber;
  }

  public void concurrentExecution(Runnable r)
  {
     try {
       synchronized(this)
       {
         if(++active != needed) wait();
```

```
            else notifyAll();
          }

          r.run();

          synchronized(this)
          {
            if(++finished != needed) wait();
            else notifyAll();
          }
        }
        catch(InterruptedException ie) {}
      }

      private final int needed;
      private int active, finished;
    }
```

Part (iii) [10 marks]

Unseen problem

```
public class RunConcurrent
{
  public RunConcurrent(int maximumNumber)
  {
     active = 0;
     waiting = 0;
     needed = maximumNumber;
     newAction = true;
  }

  public void concurrentExecution(Runnable r)
  {
    try {
     synchronized(this)
     {
       while(!newAction)
       {
          wait();
       }
       waiting++;
       while(waiting < needed) wait();
       if(++active == 1)  {
         notifyAll();
         newAction = false;
       }
     }
```

```
        r.run();

        synchronized(this)
        {
          if(--active != 0) wait();
          else
          {
            newAction = true;
            waiting = 0;
            notifyAll();
          }
        }
      }
      catch(InterruptedException ie) {}
      }

      private final int needed;
      private int active, waiting;
      private boolean newAction;

    }
```

Question 2 (25 marks)

Part (i) [13 marks]

Book work

ATCs are represented by AIEs. Generally, AIEa are considered to be a form of exceptions. So at one level it is integrated. (1 mark)

However, the normal Java rules to not apply because (2 marks each):

* Only the naming of the AsynchronouslyInterruptedException class in a throw clause indicates the thread is interruptible. It is not possible to use the name of a subclass. Consequently, catch clauses for AIEs must name the class AsynchronouslyInterruptedException explicitly and not a subclass. This is to allow AI-methods to be readily identified in the source code.

* Handlers for AsynchronouslyInterruptedExceptions do not automatically stop the propagation of the AIE. It is necessary to call the happened method in the AsynchronouslyInterruptedException class.

* Furthermore, as a result of 2) above, although catch clauses in ATC-deferred regions that name the InterruptedException or Exception classes will handle an AsynchronouslyInterruptedException this will not stop the propagation of the AIE.

* Although AsynchronouslyInterruptedException is a subclass of InterruptedException which is a subclass of Exception, catch clauses which name these classes in AI-methods will not catch an AsynchronouslyInterruptedException.

* Finally clauses that are declared in AI-methods are not executed when an ATC is thrown.

* Where a synchronous exception is propagating into an AI-method, and there is a pending AIE, the synchronous exception is lost when the AIE is thrown

Part (ii) [12 marks]

unseen problem

Method A will be terminated as the method is AIE, the finally clause will NOT be executed. (2 marks)

The AIE will be caught by the exception handler in B but remain pending, the finally clause WILL be executed. Consequently, "caught in method B" and "leaving method B" will be printed. (2 marks)

Model Answers

The AIE will be re-thrown on return from B is method C. C will be terminated immediately. Neither the exception handler nor the finally clause will be executed. (2 marks)

The AIE will propagate through D, the exception handler in D will NOT run the finally clause will NOT be executed. (2 marks)

The AIE will be caught by the exception handler in E, the finally clause WILL be executed. Consequently, "caught in method E" and "leaving method E" will be printed. The exception is still pending as no one has called happened yet. (2 marks)

Control will return to method F, and "methodF finishing" will be printed. (2 marks)

Question 3 (25 marks)

Part (i) [8 marks]

book work

From Memory	to heap (0.5 marks each)	to immortal (0.5 markseach)	to scope (1 mark each)
heap	allowed	allowed	forbidden
immortal	allowed	allowed	forbidden
scoped	allowed	allowed	allowed if same or outer scope disallowed in inner scope
local variable	allowed	allowed	allowed if same or outer scope disallowed in inner scope

Part (ii) [5 marks]

The real-time JVM will need to keep track of the currently active memory areas of each schedulable object. One way this can be achieved is via a stack. Every time a schedulable object enters a memory area, the identity of that area is pushed onto the stack. When it leaves the memory area, the identity is popped off the stack. The stack can be used to check for invalid memory assignment to and from scoped memory areas. An assignment from a scoped memory area to another scoped memory area below it in the stack is allowed. An assignment from a scoped memory area to another scoped memory area above it in the stack is forbidden. The memory assignment rules by themselves are still inadequate to avoid the dangling reference problem. particularly is a schedulable object enters the same scoped memory area twice.

To avoid this problem, the RTSJ requires that each scoped memory area has a single parent. The parent of an active scoped memory area is (in the single stack case) as follows.

* If the memory is the first scoped area on the stack, its parent is termed the primordial scope area.

* For all other scoped memory areas, the parent is the first scoped area below it on the stack.

Part (iii) [6 marks]

Unseen problem (0.5 marks each)

```
On call to nested:  stack = (heap); top = heap
```

```
On entry to r1.run:  stack = (heap-> memA); top = memA
```

```
On entry to r2.run:  stack = (heap-> memA -> memB); top = memB

On entry to r3.run:  stack = (heap-> memA -> memB -> memC);
top = memC

On entry to r4.run:  stack = (heap-> memA -> memB -> memC);
top = memA

On entry to r5.run:  stack = (heap-> memA -> memB -> memC -
> memD); top = memD

On return to r5.run:  stack = (heap-> memA -> memB -> memC -
> memD); top = memD

On return to r4.run:  stack = (heap-> memA -> memB -> memC);
top = memA

On return to r3.run:  stack = (heap-> memA -> memB -> memC);
top = memC

On return to r2.run:  stack = (heap-> memA -> memB ); top =
memB

On return to r1.run:  stack = (heap-> memA -> memB ); top =
memB

On return to nested:  stack = (heap); top = heap
```

Part (iv) [6 marks]

Unseen problem.

An exception would be raised ScopedMemoryCycle as memA would have two parents.