SOLUTION NOTES

Concurrent Systems and Applications 2002 Paper 4 Question 1 (TLH)

For each of these problems neither option (*i*) nor option (*ii*) is a universally 'right' answer and so solutions should identify a number of pros and cons of each.

(*a*) The first solution suggests adding a field, say `public int x`, to the class. The `public` modifier makes it accessible from other classes. The second solution suggests using methods for external access and using the `private` modifier to limit direct access to within the defining class:

```
private int x;

public int getX() { return x; }

public void setX(int x) { this.x = x; }
```

The first solution is evidently simpler to write. It may run faster too. The second solution, using encapsulation, can aid maintainability – e.g. the code in the `getX` and `setX` methods can be updated if the data is to be held in some other form. The second solution would have to be used if the data is to be accessed remotely through RMI.

(*b*) Suppose that the existing code is in a method `m` defined on `I1` and that its counterpart on `I2` is `n`. The first solution proposes:

```
class D extends C implements I2 {
  public void n() {
    <code before>
    m();
    <code after>
  }
}
```

The second proposes:

```
class Adapter implements I2 {
  I1 ref;

  public void n() {
    <code before>
    ref.m();
```

```
    <code after>
  }
}
```

As before the first solution may be simpler to write. The class `D` supports both the methods of `I1` and of `I2` – a problem if they have any method signatures in common that should be implemented differently for each interface. The first solution only solves the problem for this one class `C` whereas the second can be used to adapt any class implementing `I1` to the interface `I2`.

(*c*) This is a problem that arose when the standard utility classes, such as `java.util.Hashtable`, were being designed. The first solution (the one chosen there) just adds the synchronized modifier to each method, for example:

```
public void synchronized insert (Object k, Object v)
```

This often performs poorly: acquiring and releasing these locks has some cost in the case of single threaded applications and simple mutual exclusion prevents even read operations from proceeding concurrently. In any case, if the data structure forms part of a larger system then that system may provide its own concurrency control (for example by only invoking operations on the hashtable after acquiring some other lock).

The second solution addresses those problems by allowing concurrency control to be managed on a per-application basis. However, the programmer using the data structure has to be aware of this for correct operation.

(*d*) Syntactically, the only difference between the two solutions is the name of the method. In each case it would be of the form

```
public void finalize () {
  <do close operations>
}
```

The body of the method may have to interact with the server, or simply invoke `close` on the TCP socket.

The two solutions differ in when this method will come to be called. In the first case it must be invoked explicitly by the application: the application must be aware of when it has finished using the connection and it is ready to be closed. This may require extra book-keeping, for example if it is being accessed by several threads. In the second case the method will be invoked automatically once the garbage collector has determined that the object is otherwise no longer accessible to the application. This automated scheme avoids the application having to track when the connection

can be closed and removes any risk of calling the method too early (i.e. while the connection is still in use). However, while potentially simpler to the programmer, the second scheme could be overly pessimistic – there is no guarantee of exactly when the finalizer will be called and so it may be delayed some time from when the object ceases to be reachable (which may itself be delayed from when the application will no longer use it).