SOLUTION NOTES

Further Java 2001 Paper 3 Question 2 (TLH)

(*a*) Describe the differences and similarities between abstract classes and interfaces in the Java Programming Language. How would you select which kind of definition to use?

- An interface contains only method signatures, not implementations. A non-abstract class implementing the interface must provide corresponding implementations.
- An abstract class may contain both abstract methods and ordinary method definitions. As with the signatures defined on an interface, an abstract method does not define the method's implementation.
- A class may implement multiple interfaces, but can only extend one superclass (whether abstract or not).
- Both entities define new kinds of Java reference type, compatible with any object whose class extends/implements the definition.
- Interfaces cannot contain non-static fields.

Use an abstract superclass when it corresponds to a thing which is being specialized to derive subclasses (e.g. `Closure` in the next part of the question) or where common behaviour can be extracted from subclasses. Use an interface when you just want to specify some way of interacting with a thing.

(*b*) By using an inner class definition or otherwise show how this example could be re-written as a valid Java program.

One approach: define `Closure` to be an abstract superclass with an `apply()` method:

```
public class Closure {
  public abstract void apply();
}
```

Extend this in an inner class definition:

```
Closure myCounter (int start) {
  class CounterClosure extends Closure {
    int counter;
    public void apply () {
```

```
      System.out.println (counter ++);
    }
  }

  CounterClosure result = new CounterClosure ();
  result.counter = start;
  return result;
}
```

Take care over the handling of `count`: it must be pushed into the enclosed class.

(*c*) Describe three ways in which this problem can be resolved to produce (one or more) valid class definitions. State, with a brief justification, which you would use here.

Some options:

- Manually copy code from one or more of the superclasses, e.g. from `BinaryTree` into `AutoTree`. (Not good: prevents code re-use)
- 'Straighten' the hierarchy, e.g. so `BinaryTree` extends `Thread`. (Not good: introduces a spurious relationship between additional classes)
- Use an interface in place of one or more of the parents, e.g. `Runnable` in place of `Thread`.
- Express one of the relationships with containment, e.g. an inner-class `AutoTreeThread`. (Perhaps in this case the best answer: it reflects the separation between the passive data structure and the active thread that maintains it)