Solution notes

Concurrent Systems and Applications 2005 – Paper 4 Question 8 (JKF)

(*a*) (*i*) For any class that needs to be serializable, the programmer implements java.io.Serializable. This is a public interface with no methods and used only as a flag to indicate that the programmer confirms that the class is compatible with the serialization mechanism. Once a class implements serializable, all subclasses derived from it do so implicitly; it is not possible to un-implement Serializable. Serialization to/from a stream of bytes is provided by default implementations of two methods:

```
private void writeObject(java.io.ObjectOutputStream out)
    throws IOException

private void readObject(java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```

These two methods can be overridden to provide class-specific behaviour, usually by delegation to the default implementation preceded by or succeeded by some custom activity. The serialization is a deep-copy operation and will attempt to serialize every object in the object graph that is reachable through object references from the object that was explicitly serialized by the programmer.

It is possible to divert serialization to read/write a different object from the object that the programmer attempted to serialize—override readResolve and/or writeReplace to achieve that.

(*ii*) During the deep copy, the mechanism will throw a NotSerializableException if any of the objects reached by exploring the object graph from the fields of the class being explicitly serialized is found to not implement Serializable.

Furthermore, a ClassNotFoundException can be thrown if the de-serializing JVM is unable to find the class definition for a class that appears in the input byte stream.

java.io.IOExceptions can be thrown too and indicate that a general I/O problem occurred.

(*iii*) The transient modifier applies to fields only (not methods or constructors or classes or interfaces). Transient fields are not transferred by the serialization mechanism and are constructed at the receiving end by calling the no-arguments, public or protected, constructor for the data types of the fields.

(*iv*) Some programmers prefer Externalizable to Serializable because it offers finer-grained control over the process and is more easily controlled in a complex class hierarchy.

(*b*) The ClassLoader allows class definitions to be read in at runtime from a byte stream (but not to be written out again—this is a one-way operation). The default implementation reads from files on disk. Alternatives read from HTTP servers, for example. Subclasses can be created for custom purposes. The JVM ensures that type-safety is not defeated by internally representing each class with a tuple of pointers to the class' name and the classloader that loaded it. That ensures that two classes with the same name but loaded by different classloaders cannot become confused together and thus defeat the type-safety. This was broken in early JVMs.

(*c*) Native methods offer a mechanism to implement the body of a method in a language other than Java. They are useful for (perceived) performance reasons, and for talking to hardware devices (e.g. mobile 'phone hardware, robot control circuit boards, etc.). Also useful for linking against a 3rd party library that isn't available for Java.

(*d*) Reference objects, meaning java.lang.ref.Weak/Soft/PhantomReference, are used to keep track of objects when they fall out of use. When the only remaining reference to an object is a reference, it is eligible for collection by the garbage collector. A leaky cache can be constructed by having the items in the cache be reference objects (usually soft references). When the only references to objects in the cache are soft, they can be garbage collected but will hang around unless memory is scarce. Should the GC collect a weak/soft reference the reference will be "cleared" (get() returns null) and this is the indication that the item is no longer in the leaky cache and should be regenerated by whatever mechanism created it.

Phantom references are not cleared by the GC. The programmer can associate references with a reference queue—they will be placed on to the reference queue to indicate (typically) to a reaper thread that there is some clean-up work to be done with whatever resources the referent object was using before it ceased to be in use. This is so-called pre-mortem clean-up.