SOLUTION NOTES

Concurrent Systems and Applications 2003 Paper 4 Question 8 (TLH)

(*a*) *Lecture 6, memory management.* There are two main options. The first is to overload the `finalize` method on any class that requires clean-up operations. The garbage collector will cause an object's finalizer to be run at some point after it identifies that the object is unreachable. The finalizer may contain arbitrary code and should be prepared to execute in any possible context; it may run in parallel with the rest of the application, or the application may be paused while it runs. This limits what can be done safely in a finalizer – for instance most attempts to take out locks are prone to causing deadlock.

The second option is to use *phantom reference objects.* In this case, a phantom reference object would be created containing a reference to the object whose clean-up it will be managing. The phantom reference should be registered with a *reference queue.* The garbage collector will add the phantom reference to the queue when the object it refers to is collected. Note that this means that information needed for the clean-up must be held in the phantom reference object. The application is responsible for taking objects off the reference queue and performing the clean-up operations that are required.

Example use: sending appropriate "closedown" messages to a remote machine when an object representing the connection to it becomes unreachable.

(*b*) *Lecture 5, reflection.* Java's facilities for *reflection* or *introspection* allow a method to be called given its name as a string. There are built-in classes whose instances represent Java classes and the methods and fields that are defined on them. Invoking a method involves three steps (i) obtaining the `Class` object on which the method is defined, e.g. by invoking `Class.forName()` (ii) obtaining the `Method` object to be invoked, e.g. by using `getMethod()` on that `Class` object and (iii) calling `invoke` passing in the object on which to call the method and the parameters as an `Object[]` array.

Example use: in the implementation of server-side stub methods in an RPC system.

(*c*) *Lecture 5, serialization.* Serialization allows instances of any class that implements the `Serializable` interface to be converted automatically to/from a "serialized" representation by using the `readObject()` and `writeObject()` methods on `ObjectInputStream` and `ObjectOutputStream`. These streams can be constructed to access data in files, over network connections or simply in `byte[]` arrays. Fields marked `transient` will not be accessed by serialization. The programmer can provide application-specific methods for serialization by implementing the `Externalizable` interface instead of the `Serializable` one.

Example use: loading and saving data structures to disk, so long as compatibility with non-Java systems is not a concern.

(*d*) *Lecture 9, class loaders.* A custom *class loader* can be used to convert data in the form of a `byte[]` array into a `Class` object which can then be instantiated and used in the normal way. A class loader is defined by sub-classing `java.lang.ClassLoader` and overriding its `findClass` method. This method takes the name of the requested class and, if the classloader has a suitable definition, returns it as a `byte[]` array. Once the class loader has loaded one class `C`, by being explicitly requested for it, then the same class loader will be requested for subsequent classes that `C` depends on.

Example use: loading classes from sources other than the file system, for instance loading one supplied by a server over a network connection.

(*e*) *Lecture 9, native methods.* Java allows *native methods* to be defined externally in other languages. They are defined on Java classes by prefixing the method signature with the `native` keyword. The definition of the method is then supplied by some other means dependent on the JVM and on the language in which the method is being defined. For instance, a native method defined in C would probably be supplied in a shared library or `.dll` file. Note that calling a native method is likely to have a substantially higher run-time cost than calling an ordinary method and that accessing Java data structures through the Java Native Interface (JNI) is also much slower than accessing the same structures from Java.

Example use: interfacing with existing code that cannot readily be ported to Java.