

PJava defines the notion of persistent callbacks that are executed when a class is loaded into a session. Such a class is "registered as a restart callback class". We think that this type of callback is required to implement automatic loading of native methods for a particular class; the callback would load into the virtual machine's process the shared library which contains the required native methods.

In GemStone Smalltalk, a newer version of a class normally implements a migration method that accepts as an argument an instance of an older version. The migration is implemented by cloning an instance of the new version from the state of the old instance then executing the `become :` method which is implemented in class `Object`. If Java does not eventually implement `become :`, then it may still be possible to accomplish migration, but it would take much more programming effort to preserve the graph of objects being migrated.

### References

1. Design Issues for Persistent Java, Atkinson, Jordan, Daynès and Spense, 9 May 96 , submitted to POS-7.
2. Draft PJava Design 1.2, Atkinson, Daynès and Spense, 9 May 96.
3. GemStone Programming Guide v5.0, GemStone Systems Inc. 1996.
4. Java JDK version 1.0.2, Sun Microsystems Inc.

Java is a trademark of Sun Microsystems.

GemStone is a trademark of GemStone Systems Inc.

There are at least two approaches to optimizing the Java implementation to reduce the number of objects. One is to put complexity into the storage manager to recognize singly referenced objects and treat them specially. As a minimum one would try to cluster them with their parent to avoid an extra disk access. Further optimization could try to avoid the object header overhead. However adding lots of special cases in a low layer of the system such as the storage manager is likely to degrade performance and make garbage collection more complicated.

A second approach is to make the Java built-in arrays first class objects that are variable sized, and to add a `SmallInteger` class in Java. In the JDK 1.0.2, even without a handle table, only one fourth of the object identifier space could possibly be used since objects as a minimum must be aligned on 4 byte boundaries (the current handle table means alignment is only on 8 byte boundaries). Smalltalk systems typically use the OIDs that would represent unaligned addresses for classes with special implementations.

Implementing objects as variable sized could restrict virtual machine implementations by requiring a handle table. However, if a garbage collector is to be able to reclaim unused objects and compact the remaining live objects, there is a rather strong requirement that objects be relocatable, at least in a disk-based implementation.

### **Class versioning in a persistent Java**

A production database with many gigabytes of persistent objects is a long-lived unit of storage. It is frequently necessary to make changes to the applications over the life of such a database. Often this means changing a class definition while preserving existing instances of the class. In Smalltalk, we support adding, changing or removing methods from a class without requiring that a new version be created. Changing the fields defined for an instance requires creating a new version of a class and keeping the previous version alive until instances can be migrated to the new version. Instance migration must usually be deferred and done incrementally or under application control to limit impact on response times and to limit the scope of concurrency conflicts.

GemStone Smalltalk currently allows multiple versions of a class to be present in the database. Each version has a different OID and instances refer to their class by OID. References to a class within a method may be by identity (the literal variable token) or may be programmed as an explicit by-name reference ( `resolveSymbol: #aClassName` ) which always refers to the current version of the class. The literal variable references bind by OID to an Association in the name space. It is possible at the time of creating a new version of a class to control whether the new version or the old version has a constant identity for it's Association in the name space.

In the Java domain there is the need to support importing classes from external `.class` files or `.java` files. It would be desirable that all inter-class references within a loaded package be consistent. This would require that references to a class from within methods be bound to the identity of the class at the time of loading a package. In the JDK 1.0.2 inter-class references are by name lookup in the run-time class cache.



The single object space application needs to have separation between the commit or abort of a transaction, and the objects and threads that implement the user interface. A transaction abort to the persistent store should not cause unexpected changes to windows or panes of the user interface. As a minimum, objects representing the state of the user interface need to be unaffected by any rollback. It is also desirable that threads used to process user input such as mouse events are never disturbed by a transaction abort.

A third requirement for mapping threads to transactions affects companies working under the terms of Sun's commercial source code license for Java. All Java implementations must run the existing Java single-user conformance tests. So a persistent Java implementation must support multiple Java threads running in a single transaction workspace. We believe that this requirement is consistent with the requirements of the two application architectures listed above.

Our current implementation strategy is that the transaction model is primarily implemented by the storage manager. This has less flexibility than the `UpdateBookkeeper` in PJava, but has thus far been sufficient to meet the needs of our commercial customers. Our storage manager will support multiple Java threads running within one transaction workspace. When a thread requests a transaction commit, other threads in the workspace must as a minimum reach a yield point before the commit can complete, since the final stages of the commit processing require that objects within the workspace not be changing state. A more restrictive model is to consider threads within the workspace to be organized hierarchically. The workspace would be initially created owned by a specific thread, as a result of spawning a transactional thread from the RMI layer. The outer thread in the workspace could spawn additional threads, but the transaction commit would wait for the child threads to terminate with a commit or rollback status.

### **Relationship of the `transient` property to transaction semantics**

It is very desirable that transient objects, such as those implementing user interface state, not become persistent. The `transient` property of Java fields could be of use in meeting this goal. Our current implementation of transient fields causes references through a `transient` field to be ignored by a transaction commit operation, and those fields are always `null` in the persistent store.

It could also be useful to declare that instances of a class are transient, or to declare an individual object transient at creation. Declaring objects to be transient makes it possible for the transaction commit operation and/or the store barrier in the virtual machine to throw an exception if an object declared transient would become persistent at commit. By defining such references to be a programming error, the Java virtual machine could ensure that certain transient objects remained transient. Accomplishing this would require changes to packages such as `java.io`, `java.net`, and `java.awt`, and might yield improved reliability for persistent Java applications.

## **Persistent Java issues from a GemStone perspective**

Allen Otis, R.Venkatesh  
GemStone Systems Inc.  
5 September, 1996

This workshop submission is offered based on the authors' experiences in developing and using GemStone, a multi-user persistent Smalltalk system. The authors are currently developing Java server products based on GemStone's storage manager technology and using Java under terms of Sun's commercial Java source code license.

### **Threads and transactions**

Two possible architectures for an OLTP application are client-server and single object space. We will attempt to give a brief description of each architecture and its requirements for mapping threads to transactions.

In a client-server application, the user interface portion of the application executes in a non-transactional runtime environment in a client process. The actual transaction begin, update and commit or abort operations that will affect a persistent store are executed in a separate server process. The client could be a Java program communicating with a persistent Java server process via a remote messaging interface. In such a system, regardless of whether the server is instructed by the client to commit or abort, the server is expected to service the next client request to start the next transaction. The thread(s) in the server process handling communications with clients needs to run continuously and there should be no need for a server process to restart communication threads after a transaction abort. In a Java system this communication with the client would be implemented with Sun's Remote Message Interface (RMI) package. This would suggest that the server main program and some of the threads invoked from it need to be non-transactional. At the very least, the objects representing communication state should never be affected by a rollback operation. In a large scale system, it is desirable to support multiple clients per server process. So there needs to be concurrent execution of multiple threads each representing an outer-level transaction in the server.

In a single object space application, both the user interface and the transactions modifying the persistent store would run in one persistent Java process. Earlier versions of GemStone offered a product named "GeODE" using this architecture, which was used for complex decision support applications that used an X-terminal. The single object space architecture is only feasible on a high bandwidth local area network with low latency, since either X-window operations or workstation disk reads are sent over the network to the server. Most commercial GemStone applications are using the client-server architecture to reduce memory requirements on the client machine or cope with slower networks. However, a single object space application is significantly easier to program than client-server for many applications. Thus the single object space architecture might be very desirable for persistent Java applications executing against a personal or departmental database, particularly in research environments.