



Object Design's PSE for Java

**Gordon Landis
Object Design, Inc.**



Guiding Principles

- **Tight language binding**
 - must support language features (principle of orthogonality)
 - persistence should be transparent
- **Support for multiple storage managers**
 - PSE at the low end, ObjectStore at the high end



Persistent interface to Java Requirements: Language Considerations

- Portable
 - 100% Java
 - should not require specialized or modified environment or VM
- Garbage collected
 - should support persistence by reachability
 - should not interfere with the GC of transient objects
 - should support GC on persistent database objects (not yet released)
- Multi-threaded
 - should allow different threads to cooperate in the same transaction
 - should allow different threads to operate in independent transactions, contending for concurrent lock access (currently in PSE Pro only)



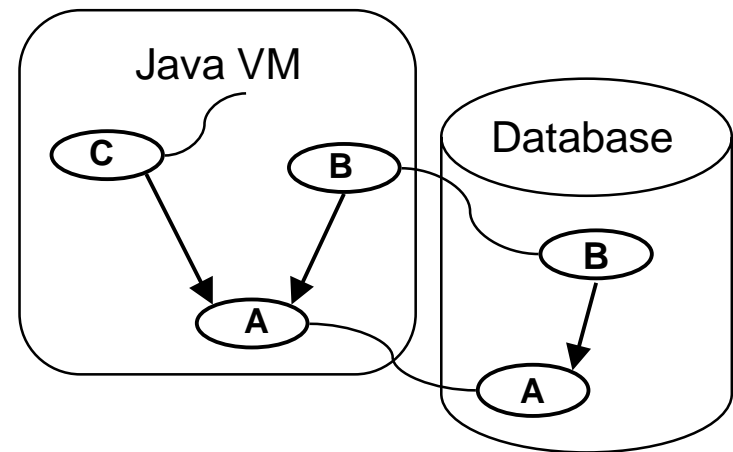
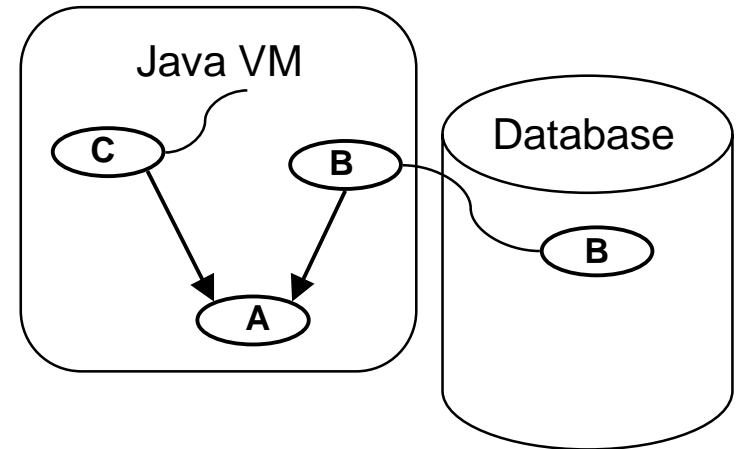
PSE and OSJI Considerations

- PSE and OSJI share the same front end and the same API (the OSJI API is a superset of the PSE API). This was a design goal motivated by our “PSE ubiquity” strategy.
- The front end classes bind to the appropriate back end (storage subsystem) based on
 - the **COM.odi.env** Java system property
 - the **CLASSPATH** environment variable
- Code (both Java source, and compiled classfile byte-code) are compatible between PSE\ and OSJI (if you use only the common subset API);
- Databases are **NOT** compatible between PSE and OSJI.



Reachability

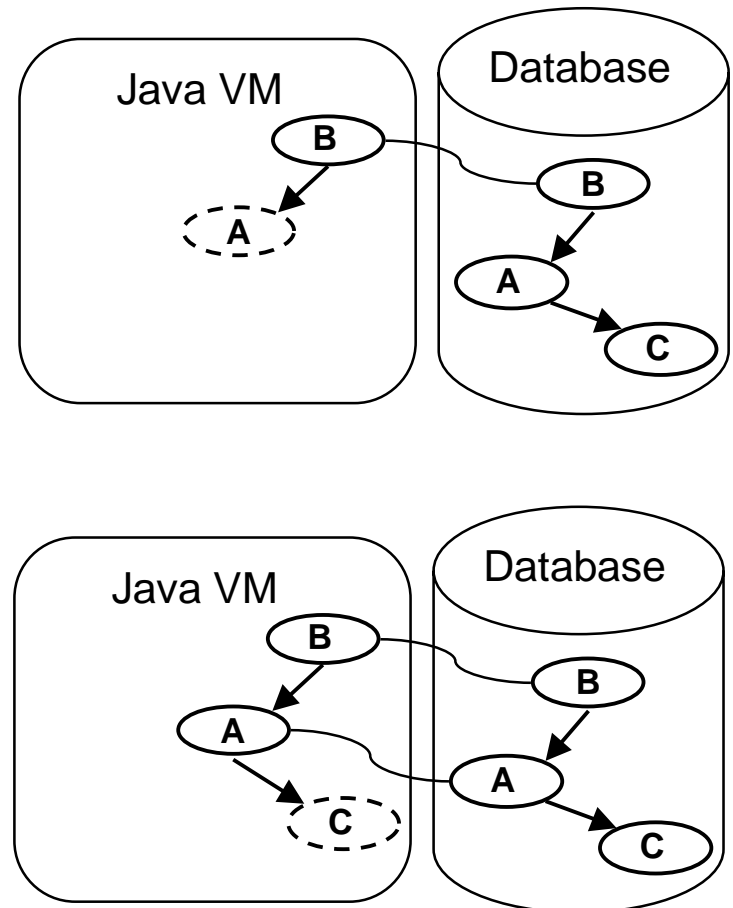
- Object B exists in the database
- Object C is any transient object
- Object A is a new persistent capable object:
`A = new MyClass(...);`
- Object B stores a reference to object A:
`txn = Transaction.begin();`
`B.myRef = A;`
- Object C not reachable - remains transient
- Object A is reachable - Migrated to database at commit time:
`txn.commit();`





Transparent Object “Faulting”

- Object *B* is *active*
- Object *A* is referenced by *B* but is *hollow*
- Persistent images of *A* and *B* are managed by the internal storage engine:
 - May or may not be in cache memory
 - Automatically retrieved if needed
- Object *B* calls a method on *A*
 - Triggers *fetch* primitive
- Object *A* is made active:
 - Data for *A* retrieved from database
 - Referenced object *C* materialized in Java VM
 - Existing *C* used if there is one
 - Hollow *C* created otherwise





Development Process

- **Code Person.java source files**
- **Compile with *javac* command:**
`javac Person.java`
- **Postprocess with *osjcfp* command:**
`osjcfp -dest osjdir @cfpargs`
 - Alternatively, manually annotate code with `fetch()`, `dirty()` calls
- **Debug with Java debugger**
 - Availability and use depends on environment
- **Define CLASSPATH to select persistified classes**



Post processing with OSJCFP

- Provides the meta-data (schema) information
- Annotates classes and their methods
 - Persistence-capable classes inherit from Persistent (in the future, they will simply *implement* the *IPersistent interface*)
 - this provides the “barrier” flags -- data members that indicate the state of the object: readlocked, writelocked, etc (more on this later)
 - Persistence-aware class methods (including persistence-capable class methods) have `fetch()` and `dirty()` calls inserted
 - ***This is the analog to page-faulting in ObjectStore C++ interface***
 - these calls check the “barrier” flags, mediating all access to persistent data
 - normal optimizations: *this* arg and loop variables (can be dangerous in some transaction scenarios)



Transient versus Persistence-Capable Versions of Classes

- **Instances of Transient classes cannot be stored in the database**
- **Persistence-Capable versions are annotated to support database storage access**
 - Embedded *fetch* and *dirty* hooks; inheritance from Persistent
 - Extended schema information, including size and shape information for object and arrays, type and name for each field -- this is stored in the xxxClassInfo.class file
- **Instances of Persistence-Capable versions *need not* be persistent (“persistence orthogonal to type”)**
 - Persistence determined at runtime
 - Persistent and transient instances coexist and share methods
- **Array, String, and primitive wrapper classes are automatically persistence-capable**



Persistence Capable vs Aware

- **Persistence Capable:** A class or object that can be stored in the database if it is reachable from a persistent object or root.
 - Necessary *fetch* and *dirty* calls added by osjcfp (or manual annotation)
 - Any class that is persistence-capable is automatically persistence-aware as well
- **Persistence Aware:** A class or object that can operate on persistent data, but cannot be stored in a database
 - Relevant *only* if class may access non-private data members of a persistent class



OSHashtable and OSVector

- **Java provides Hashtable and Vector classes, but we have created OSHashtable and OSVector classes**
 - just like Hashtable and Vector, except persistence-capable
 - we could not simply change the existing Hashtable and Vector in place, because these are used all through the VM (and PSE) internals
 - performance penalty
 - recursion and breakage
 - re-implemented for scalability reasons



Creating Persistent Objects

- **First create a new Java object:**
 - Standard Java *new* operator (no special overloads as in C++ interface)
- **Object may become persistent in three ways:**
 - Explicitly migrating into a database
 - Defining the object as a database root
 - By “reachability”
- **All operations that create objects require:**
 - transaction be started in Update mode
 - database open for update



Object Materialization States

- **Hollow object (Read Barrier and Write Barrier both up)**
 - An object that acts as a handle to a persistent object in a database
 - If you access a hollow object its contents are fetched automatically
- **Active object (Read Barrier and/or Write Barrier down)**
 - An object which has been fetched and has values in it
 - Active states include Read access and Update access
- **Stale object**
 - A persistent object that cannot be accessed
 - Any attempt to use this object will raise an exception
 - Object Identity has been lost



Object Identity

- **The ObjectTable contains ref to every persistent object that has been materialized in the Java VM (hollow or active)**
- **Once an object is made “stale” its reference is removed from the OT (object identity is lost, any use of the ref fails)**
- **The OT normally uses weak references, so it does not impede the Java (transient) GC**
 - a “dirty” object gets a strong reference, so that we don’t lose the changes to a zealous GC
 - weak references are broken in some JVMs -- you can turn them off (downside: GC is impeded)
- **Strings and primitive wrappers do not have OT references**
 - we forego object identity
 - this is reasonable, because they are immutable



Transactions

- **A transaction can be specified as update or read-only**
 - read-only transactions prevent modification of persistent data
- **Transactions are “dynamic” in scope**
 - Program must explicitly define start and end
 - In general, `Transaction.begin()` and `Transaction.commit` are invoked in the same top-level method.
- **Objects are always up to date within the transaction**
 - When transaction ends, other users can change data
 - Access outside a transaction may be based on out-of-date data
- **Currently no support for MVCC, nested transactions, or checkpoint/refresh**



Commit

The following activities occur during commit:

- **Transient objects referenced by persistent objects become persistent**
- **Modified persistent objects are flushed to the database**
- **“Tombstones” are written for deleted objects**
- **Object state is *downgraded* to specified *retain* parameter**
 - E.g. RETAIN_HOLLOW causes objects in Read or Write state to be made hollow, but stale objects are still stale
- **The ObjectStore client releases locks**
 - Lazy lock release if no other clients are waiting (controlled by cache manager)
 - Any waiting clients automatically resume processing



Transactions and Object Materialization

- **Object Materialization may be downgraded by Commit, Abort, or Evict operation, and any RETAIN parameter:**
 - stale, hollow, read, write
- **Within a transaction materialization may automatically upgrade:**
 - Read method: Hollow-->Read state
 - Write method: Hollow-->Write or Read-->Write
- **Between transactions**
 - No automatic upgrade
 - Objects may be left in Read or Update state by prior commit.
 - Access to Stale or Hollow objects will raise an exception.



Commit Retain Options

commit (ObjectStore.RETAIN_STALE) - default option

- Aggressive GC, loses identity
- Important workaround for lack of weak references in JDK 1.0

commit (ObjectStore.RETAIN_HOLLOW)

- Preferred choice - allows dynamic GC but retains identity
- Object references released, allowing garbage collection
- Any access of object triggers fetch, refreshes data

commit (ObjectStore.RETAIN_READ)

- Allows access outside txn, reduces object access cost
- Makes it possible to invoke a method on a persistent object even if a transaction is not in progress

commit (ObjectStore.RETAIN_WRITE)

- Allows update outside txn; not generally recommended
- Updates outside transaction are lost in next transaction



Retaining Objects Across Transactions

- The RETAIN mode controls the accessibility of objects following a transaction commit or abort
- In PSE, objects that are readable at the end of a transaction *continue* to be readable at the start of the next transaction



Abort

The following activities occur during abort:

- **Transient objects that were made reachable from persistent objects by the transaction are left transient**
- **All persistent objects in the Java VM are marked stale**
 - unless specified by `Transaction.setDefaultAbortRetain()`
- **Objects deleted or modified within the transaction are rolled back with their original data**
- **All database locks are released**
- **Transient objects are untouched:**
 - The program must ensure they are consistent with the rolled-back persistent objects



Summary

- **Transparent persistence interface implemented via post-processor**
- **100% pure Java implementation means that PSE runs on any JVM**
- **Object identity managed through Object Table that interacts well with GC (using weak references)**
- **Extensions to the normal “serializable” transaction semantics allow access to previously read information even outside of any transaction**
- **Support for multiple storage systems means that there is an easy upgrade path from free PSE product to high-end ObjectStore database system**