

[2] A. Malhotra and S.J. Munroe, *Support for Persistent Objects: Two Architectures*, Proc. 25th Hawaii Intl. Conf. on System Sciences, 1992, pp. 737-746.

[3] A. Malhotra and S.J. Munroe, *Schema Evolution in Persistent Object Systems*, Proc. Seventh Intl. Workshop on Persistent Object Systems, Cape May, NJ, 1996.

Extents

We propose two new, optional, keywords `extent` and `fullExtent` that can be used to name collections of all instances of a class and all instances of a class and all its subclasses.. Separate collections are maintained for each store and can be accessed as `ExtentName.StoreName` and `FullExtentName.StoreName`. They can be used to query all instances of a class or all instances of a class meeting certain conditions, etc.

Class Versions

To support schema evolution, we propose that classes be able to support versions [3]. Each time a class is changed a new version is created. Instances of several versions of a class can exist concurrently. Versions of a class can have different attributes and different methods as well as different implementations of the same method. Since each version has its own method table, identically named method calls on instances of different versions can be routed to different implementations.

To support versions we need a new class called `Version`. Each `Class` instance owns a sequence of versions. The `newInstance` method of `Class` creates an instance of the latest version. For compatibility it is sometimes necessary to create objects of earlier versions. This requires a `newInstance` method for `Class` that takes a version as an argument.

```
newInstance(Version);
```

and similar variants of the other new methods.

We also need to be able to get the latest version of a `Class`

```
Version latestVersion();
```

and all versions of a `Class`

```
Version[] allVersions();
```

and the latest version at a particular time

```
Version latestVersion(Date date, Time time);
```

For `Object` we need to add a method, similar to `getClass`, to get its version

```
Version getVersion();
```

To change an object instance from one version to another, we need a `morph` method

```
morph(Version);
```

This converts the object from its current version to the argument version. These methods have to be written by the user using base facilities supplied by the system. `Morph` methods may not exist from all versions to all versions.

References

- [1] M.P. Atkinson, M.J. Jordan, L. Daynes, S. Spence, *Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system*, Proc. Seventh Intl. Workshop on Persistent Object Systems, Cape May, NJ, 1996.

```

        // creates new heap
boolean isPersistent();
boolean isCollectible();
boolean isAuditable();
Class[] allClasses();
        // return all classes in heap
RegisterRoot(String name, Object obj);
        // make obj a persistent root in the heap
DiscardRoot(String name);
        // discard the named root from the roots of the heap
}

```

Persistence Summary

If the heap is **transient**, object instances can be created with `new` (if it is the default heap), `newIn` and `newNear`. All instances in the heap disappear when the program ends.

If the heap is **persistent with garbage collection**, object instances can be created with `new` (if it is the default heap), `newIn` and `newNear`. Instances reachable from persistent roots are stabilized when the program ends normally or if the top level transaction commits.

If the heap is **persistent without garbage collection**, transient object instances can be created with `new` (if it is the default heap), `newIn` and `newNear`. These instances all disappear when the program ends. Persistent object instances can be created with `newPerm` (if it is the default heap), `newPermanentIn` and `newPermanentNear`. These instances are stabilized when the program ends normally or if the top level transaction commits provided a `delete` has not been invoked on them.

Alternate Proposal

Objects in all heaps are created with `new`, `newIn` and `newNear`.

If the heap is **transient**, all instances in the heap disappear when the program ends.

If the heap is **persistent with garbage collection**, instances reachable from persistent roots are stabilized when the program ends normally or if the top level transaction commits.

If the heap is **persistent without garbage collection**, all instances created in it are stabilized when the program ends normally or if the top level transaction commits provided that a `delete` method has not been executed on them.

A program that works only with transient objects just uses the default heap and creates objects with `new`. If the style of the default heap is changed to persistent with garbage collection the program still creates objects with `new` and we get persistence by reachability. If the program wants to work with persistent objects that are not garbage collected it must have at least two heaps, a transient heap for transient objects and a persistent heap for persistent objects. It creates transient objects in the default transient heap with `new` and creates persistent objects in the persistent heap with `newIn` or `newNear`. If there is only one persistent heap the `newPerm` operation can be used as a synonym for `newIn(pHeap)`.

This is a somewhat simpler proposal but has the disadvantage that in the case of persistence without garbage collection transient objects cannot be created in the persistent heap and so the program must work with at least two heaps.

- for non-collectible heaps, all objects created by `newPermanentInstance`, `newPermanentInstanceIn` or `newPermanentInstanceNear` for which a `delete` method was not executed are stabilized.

Stabilized objects are written to permanent storage and are recoverable in case of software or hardware failure. As a practical matter, only changed objects are written to permanent storage.

For collectible heaps, there is a difficult problem in deciding which persistent roots to start from. Starting from persistent roots in all reachable stores is clearly infeasible. Starting from only the persistent roots in stores that the program has opened may lead to deletion of objects that were not reachable from roots in opened stores but could be reached from roots in other unopened stores. We recommend that reachability only trace objects from persistent roots in opened stores. It is the user's responsibility to make sure all relevant stores are opened.

Reachability does not trace pointers from collectible heaps to non-collectible heaps -- there is no need to do this. Nor does it trace pointers from non-collectible heaps into collectible heaps. This can lead to dangling references but the situation is no different from standard databases and can be mitigated by referential integrity and other techniques.

The Store and Heap Classes

```
class Store {
    String Name;
    Path StorePath;
    Collection Heaps;
    Store(String name, Path path,...);
        // create new store
    static Store Find(String name, Path path,...);
        // find store on (possibly remote) path
    boolean Open(Store store, AccessRights rights);
        // open the store with requested rights
    boolean Close(Store store);
        // close the given store
    Heap getHeap(String name);
        // return the Heap with the given name
    Heap[] allHeaps();
        // return all heaps in the store
}

class Heap {
    String Name;
    Store Parent;
    boolean Persistent;
    boolean Collectible;
    boolean Auditable;
    Heap(String name, Store store,
        boolean persist, boolean collect, boolean audit);
```

Creating Objects

Objects created with the `newInstance` method of `Class` (corresponding to `new`) are created in the default heap. This heap is initialized when the program is loaded and may be transient and garbage collected or it may be persistent and may or may not be garbage collected. The characteristics of the default heap are determined by a initialization file or by a command line argument.

Objects created in a persistent heap with garbage collection become permanent if they are reachable from a persistent root and are not garbage collected. This provides persistence by reachability.

If the default heap is persistent without garbage collection, objects created with `new` are transient and disappear when the program ends. Objects created in a default persistent heap that is not garbage collected are made permanent only if they are created with an additional method, `newInstancePermanent`, of `Class` (corresponding to `newPerm`).

We propose four additional methods of `Class` which allow objects to be created in specific heaps.

```
newInstanceIn(Heap);
```

creates instances in garbage collected heaps and transient instances in persistent non-garbage collected heaps.

```
newInstancePermanentIn(Heap);
```

creates permanent instances in persistent non-garbage collected heaps. The `newInstance` and `newInstancePermanent` methods call `newInstanceIn` in the default heap as appropriate. Similarly,

```
newInstanceNear(Object);
```

```
newInstancePermanentNear(Object);
```

are advisory methods which create object instances, if possible, in the same heap and, if possible, on the same page as the argument object.

Object Deletion

In persistent garbage collected heaps, objects that are not reachable from a persistent root are eventually garbage collected. In persistent non-garbage collected heaps, objects are deleted only if the `delete` method is invoked on them. `delete` is a method of `Object`.

```
delete();
```

In auditable heaps, the `delete` method causes an audit trail record to be written out with details of who deleted the object, at what time, etc.

Object Persistence

Using the above methods, transient and persistent instances can be created for any class.

When the program ends, all objects created in transient heaps disappear. For persistent heaps, if the program ends by not being able to handle an exception, or in the transactional case, the top-level transaction aborts, there are no changes to any objects accessed by the program. If the program ends successfully, or in the transactional case, if the top-level transaction commits:

- for collectible heaps, all objects reachable from a persistent root are stabilized.

Persistent Java Objects: A Proposal

Ashok Malhotra

IBM Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598
malhotra @ vnet.ibm.com 914-784-6182

Motivation

The database community talks about commercial databases in the petabytes. As databases start to store large objects such as X-Ray images and video clips this is certainly not inconceivable. We feel that garbage collection and persistence by reachability [1] are not practical over such large databases with current technology. Also, Java applications must be able to work with a variety of stores each of which may offer their own style of persistence. Thus, while persistence by reachability offers significant usability advantages, we feel that it cannot be the only persistence mechanism for Java. This proposal combines persistence by reachability with persistence using explicit deletes. The differentiation is by heap, which allows the two mechanisms to be combined in a flexible manner.

Stores and Heaps

Heaps

Java programs can create objects in several heaps. Heaps are first class objects and have various characteristics. These characteristics influence the management of objects that are created in them. Simple Java programs work with default heaps and do not need any changes to the programming model.

Stores

Stores correspond to physical partitions of available persistent space and contain heaps. In the case of Virtual Memory implementations with long addresses [2], stores correspond to a range of addresses. In other architectures they may correspond to physical or logical partitioning of disk or other permanent memory space. Stores can be relocated from one machine to another. A Java program can access objects in several stores some of which may be on the machine it is running in and others on other machines.

Stores must be opened before they can be used. The `Open` method checks authority and may restrict functionality based upon it.

Objects in one store generally contain pointers to objects in the same store. Pointers from one store to another are possible, but, for reasons that become clear later, should be minimized.