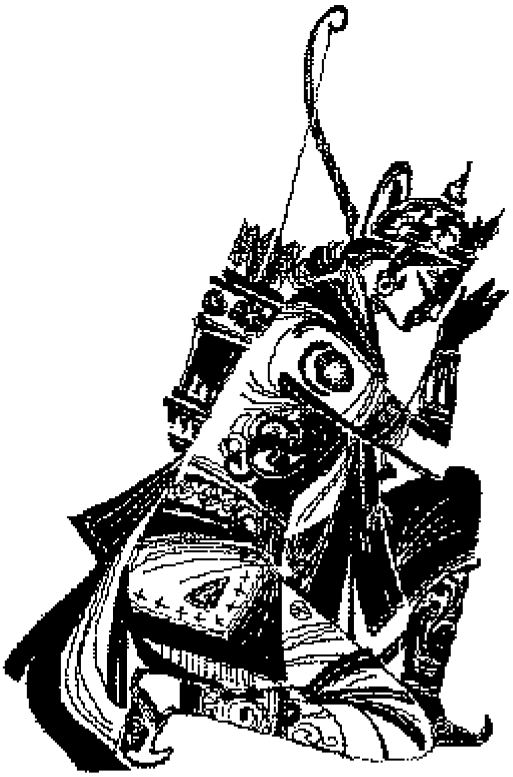

JavaArjuna

Programmer's Guide

Table of Contents

Manual structure

The structure of this manual is as follows:

Chapter 1. Advanced programming: A description of the *JavaArjuna API*, a high-level API for constructing transactional applications using objects which are persistent and concurrency controlled.

Chapter 2. Constructing a *JavaArjuna* application: A tutorial on writing applications using the *JavaArjuna* API.

Chapter 3. Useful hints and tips: Some guidelines and recommendations that should be followed when writing applications; including some hints for debugging such applications.

Appendix A. User classes: A quick reference to those *JavaArjuna* classes that a programmer will most typically use.

1. Progamming with JavaArjuna

This chapter contains a description of the use of the advanced *JavaArjuna* classes and facilities. The classes mentioned in this chapter are the key to writing fault-tolerant applications with *JavaArjuna*. Thus, after describing them we shall apply them in the construction of a simple application. The classes to be described in this chapter can be found in the `com.arjuna.JavaArjuna`, `com.arjuna.JavaArjuna.ClassLib` and related packages.

In keeping with the object-oriented view, the mechanisms needed to construct reliable distributed applications are presented to programmers in an object-oriented manner. Some mechanisms need to be inherited, for example, concurrency control and state management; while other mechanisms, such as object storage and transactions, are implemented as *JavaArjuna* objects that are created and manipulated like any other object.

In the following section we shall present a brief overview of the *JavaArjuna* system, identifying the major system components and how they interact. When the manual talks about using persistence and concurrency control facilities it assumes that the advanced *JavaArjuna* classes are being used. If this is not the case then the programmer is responsible for all of these issues. Section 2.2 will describe *JavaArjuna* in more detail.

1.1 *JavaArjuna* overview

JavaArjuna exploits object-oriented techniques to present programmers with a toolkit of Java classes from which application classes can inherit to obtain desired properties, such as persistence and concurrency control. These classes form a hierarchy, part of which is shown below.

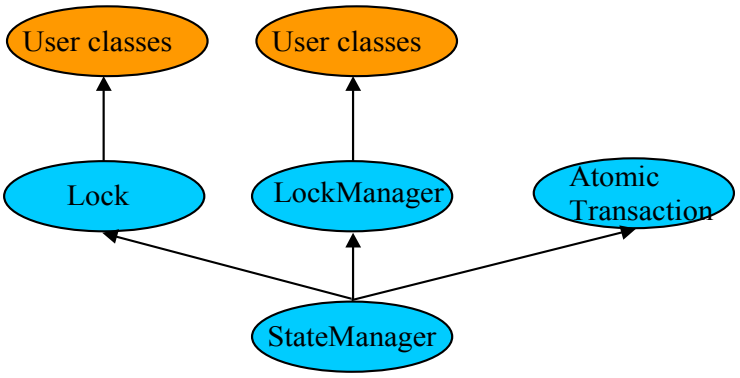

Figure 1: JavaArjuna class hierarchy

Apart from specifying the scopes of transactions, and setting appropriate locks within objects, the application programmer does not have any other responsibilities: *JavaArjuna* guarantees that transactional objects will be registered with, and be driven by, the appropriate transactions, and crash recovery mechanisms are invoked automatically in the event of failures.

1.1.1 Saving object states

JavaArjuna needs to be able to remember the state of an object for several purposes, including recovery (the state represents some past state of the object) and persistence (the state represents the final state of an object at application termination). Since these requirements have common functionality they are all implemented using the same mechanism: the classes `InputObjectState` and `OutputObjectState`. The classes maintain an internal array into which instances of the standard types can be contiguously packed (unpacked) using appropriate `pack (unpack)` operations. This buffer is automatically resized as required should it have insufficient space. The instances are all stored in the buffer in a standard form (so-called network byte order) to make them machine independent. Any other architecture independent format (such as XDR or ASN.1) could be implemented simply by replacing the operations with ones appropriate to the encoding required.

1.1.2 The object store

Implementations of persistence can be affected by restrictions imposed by the Java SecurityManager. Therefore, the object store provided with *JavaArjuna* is implemented using the techniques of interface/implementation separation described earlier. The current distribution has implementations which write object states to the local file system or database, and remote implementations, where the interface uses a client stub (proxy) to remote services.

Persistent objects are assigned unique identifiers (instances of the `Uid` class), when the are created, and this is used to identify them within the object store. States are read using the `read_committed` operation and written by the `write_(un)committed` operations.

1.1.3 Recovery and persistence

At the root of the class hierarchy is the class `StateManager`. This class is responsible for object activation and deactivation and object recovery. The simplified signature of the class is:

```
public abstract class StateManager
{
public boolean activate ();
public boolean deactivate (boolean commit);

public Uid get_uid (); // object's identifier.

// methods to be provided by a derived class

public boolean restore_state (InputObjectState os);
public boolean save_state (OutputObjectState os);

protected StateManager ();
protected StateManager (Uid id);
};
```

Objects are assumed to be of three possible flavours. They may simply be *recoverable*, in which case `StateManager` will attempt to generate and maintain appropriate recovery information for the object. Such objects have lifetimes that do not exceed the application program that creates them. Objects may be *recoverable and persistent*, in which case the lifetime of the object is assumed to be greater than that of the creating or accessing application, so that in addition to maintaining recovery information `StateManager` will attempt to automatically load (unload) any existing persistent state for the object by calling the `activate` (`deactivate`) operation at appropriate times. Finally, objects may possess none of these capabilities, in which case no recovery information is ever kept nor is object activation/deactivation ever automatically attempted.

If an object is recoverable (or persistent) then `StateManager` will invoke the operations `save_state` (while performing `deactivate`), and `restore_state` (while performing `activate`) at various points during the execution of the application. These operations *must* be implemented by the programmer since `StateManager` cannot detect user level state changes. (We are examining the automatic generation of default `save_state` and `restore_state` operations, allowing the programmer to override this when application specific knowledge can be used to improve efficiency.) This gives the programmer the ability to decide which parts of an object's state should be made persistent. For example, for a spreadsheet it may not be necessary to save all entries if some values can simply be recomputed. The `save_state` implementation for a class `Example` that has integer member variables called `A`, `B` and `C` could simply be:

```
public boolean save_state(OutputObjectState o)
{
    if (!super.save_state(o))
        return false;

    try
    {
        o.packInt(A);
        o.packInt(B);
        o.packInt(C));
    }
    catch (Exception e)
    {
        return false;
    }

    return true;
}
```

Note, it is necessary for all `save_state` and `restore_state` methods to call `super.save_state` and `super.restore_state`. This is to cater for improvements in the crash recovery mechanisms.

1.1.4 The life-cycle of a Transactional Object for Java

A persistent object not in use is assumed to be held in a *passive* state with its state residing in an object store and *activated* on demand. The fundamental life cycle of a persistent object in *JavaArjuna* is shown in Figure 2.

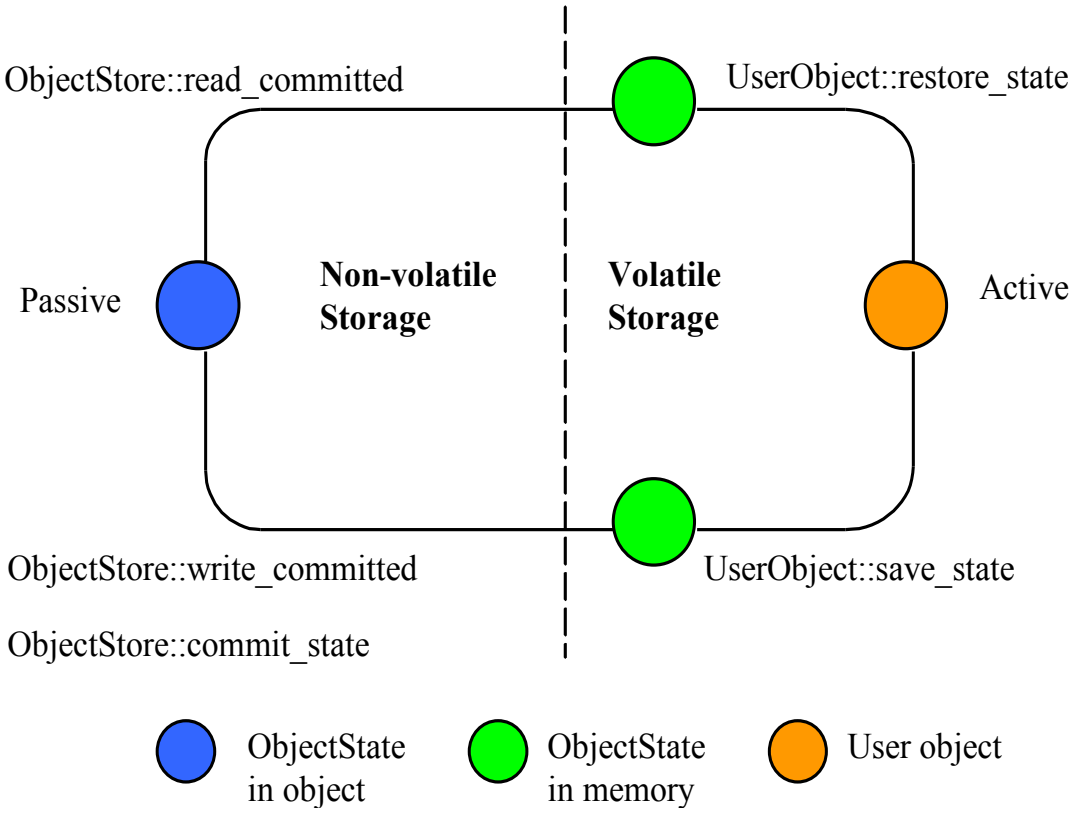

Figure 2: The life cycle of a persistent object

(i) The object is initially passive, and is stored in the object store as an instance of the class `OutputObjectState`.

(ii) When required by an application the object is automatically *activated* by reading it from the store using a `read_committed` operation and is then converted from an `InputObjectState` instance into a fully-fledged object by the `restore_state` operation of the object.

(iii) When the application has finished with the object it is *deactivated* by converting it back into an `OutputObjectState` instance using the `save_state` operation, and is then stored back into the object store as a *shadow copy* using `write_uncommitted`. This shadow copy can be committed, overwriting the previous version, using the `commit_state` operation. The existence of shadow copies is normally hidden from the programmer by the transaction system. Object de-activation normally only occurs when the top-level transaction within which the object was activated commits.

During its life time, a persistent object may be made active then passive many times.

1.1.5 The concurrency controller

The concurrency controller is implemented by the class `LockManager` which provides sensible default behaviour while allowing the programmer to override it if deemed necessary by the particular semantics of the class being programmed. As with `StateManager` and persistence, concurrency control implementations are accessed through interfaces. As well as providing access to remote services, the current implementations of concurrency control available to interfaces include:

- local disk/database implementation, where locks are made persistent by being written to the local file system or database.
- a purely local implementation, where locks are maintained within the memory of the virtual machine which created them; this implementation has better performance than when writing locks to the local disk, but objects cannot be shared between virtual machines. Importantly, it is a basic Java object with no requirements which can be affected by the SecurityManager.

The primary programmer interface to the concurrency controller is via the `setlock` operation. By default, the runtime system enforces strict two-phase locking following a multiple reader, single writer policy on a per object basis. However, as shown in Figure 1, by inheriting from the `Lock` class it is possible for programmers to provide their own lock implementations with different lock conflict rules to enable *type specific concurrency control*.

Lock acquisition is (of necessity) under programmer control, since just as `StateManager` cannot determine if an operation modifies an object, `LockManager` cannot determine if an operation requires a read or write lock. Lock release, however, is under control of the system and requires no further intervention by the programmer. This ensures that the two-phase property can be correctly maintained.

```
public abstract class LockManager extends StateManager
{
public LockResult setlock (Lock toSet, int retry, int timeout);
};
```

The `LockManager` class is primarily responsible for managing requests to set a lock on an object or to release a lock as appropriate. However, since it is derived from `StateManager`, it can also control when some of the inherited facilities are invoked. For example, `LockManager` assumes that the setting of a write lock implies that the invoking operation must be about to modify the object. This may in turn cause recovery information to be saved if the object is recoverable. In a similar fashion, successful lock acquisition causes `activate` to be invoked.

The code below shows how we may try to obtain a write lock on an object:

```
public class Example extends LockManager
{
public boolean foobar ()
{
  AtomicAction A = new AtomicAction;
  boolean result = false;

  A.begin();

  if (setlock(new Lock(LockMode.WRITE), 0) == Lock.GRANTED)
  {
    /*
     * Do some work, and JavaArjuna will
     * guarantee ACID properties.
     */

    // automatically aborts if fails

    if (A.commit() == AtomicAction.COMMITTED)
    {
      result = true;
    }
  }
  else
    A.rollback();

  return result;
}
}
```

1.1.6 The transaction protocol engine

JavaArjuna transaction protocol engine is represented by the `AtomicAction` class, which uses `StateManager` in order to record sufficient information for crash recovery mechanisms to complete the transaction in the event of failures. It has methods for starting and terminating the transaction, and, for those situations where programmers require to implement their own resources, methods for registering them with the current transaction. Because *JavaArjuna* supports subtransactions, if a transaction is begun within the scope of an already executing transaction it will automatically be nested.

The JTS specification allows both single and multi-threaded implementations. *JavaArjuna* is multi-threaded aware, allowing each thread within an application to share a transaction or execute within its own transaction. Therefore, all *JavaArjuna* classes are also thread safe.

1.1.7 Example

The simple example below illustrates the relationships between activation, termination and commitment:

```
{
. . .
O1 objct1 = new objct1(Name-A);/* (i) bind to "old" persistent object A */
O2 objct2 = new objct2();      /* create a "new" persistent object */
tx.begin();        /* (ii) start of atomic action */

objct1.op(...);               /* (iii) object activation and invocations */
objct2.op(...);
. . .
tx.commit(true);  /* (iv) tx commits & objects deactivated */
}                             /* (v) */
```

The execution of the above code involves the following sequence of activities:

(i) Creation of bindings to persistent objects; this could involve the creation of stub objects and a call to remote objects. In the above example we re-bind to an existing persistent object identified by `Name-A`, and a new persistent object. A naming system for remote objects maintains the mapping between object names and locations and is described in a later chapter.

(ii) Start of the atomic transaction.

(iii) Operation invocations: as a part of a given invocation the object implementation is responsible to ensure that it is locked in read or write mode (assuming no lock conflict), and initialised, if necessary, with the latest committed state from the object store. The first time a lock is acquired on an object within a transaction the object's state is acquired, if possible, from the object store.

(iv) Commit of the top-level action. This includes updating of the state of any modified objects in the object store.

(v) Breaking of the previously created bindings.

1.2 The *JavaArjuna* class hierarchy

The principal classes which make up the class hierarchy of *JavaArjuna* are depicted below.

```
StateManager            // Basic naming, persistence and recovery control
  LockManager           // Basic two-phase locking concurrency control
service
    User-Defined Classes
  Lock                  // Standard lock type for multiple readers/single
writer
    User-Defined Lock Classes
  AbstractRecord        // Important utility class, similar to Resource
    RecoveryRecord      // handles object recovery
    LockRecord          // handles object locking
    RecordList          // Intentions list
    other management record types
AtomicAction            // Implements transaction control abstraction
  TopLevelAction
Input/OutputBuffer // Architecture neutral representation of an objects'
state
  Input/OutputObjectState     // Convenient interface to Buffer
ObjectStore             // Interface to the object storage services
```

Figure 3: JavaArjuna class hierarchy.

Programmers of fault-tolerant applications will be primarily concerned with the classes `LockManager`, `Lock` and `AtomicAction`. Other classes important to a programmer are `Uid`, and `ObjectState`. Most *JavaArjuna* classes are derived from the base class `StateManager`, which provides primitive facilities necessary for managing persistent and recoverable objects. These facilities include support for the activation and de-activation of objects, and state-based object recovery. The class `LockManager` uses the facilities of `StateManager` and `Lock` to provide the concurrency control (two-phase locking in the current implementation) required for implementing the serialisability property of atomic actions. The implementation of atomic action facilities is supported by `AtomicAction` and `TopLevelAction`.

Most *JavaArjuna* system classes are derived from the base class `StateManager`, which provides primitive facilities necessary for managing persistent and recoverable objects. These facilities include support for the activation and de-activation of objects, and state-based object recovery. The class `LockManager` uses the facilities of `StateManager` and provides the concurrency control required for implementing the serialisability property of atomic actions.

Consider a simple example. Assume that `Example` is a user-defined persistent class suitably derived from the `LockManager`. An application containing an atomic transaction `Trans` accesses an object (called `O`) of type `Example` by invoking the operation `op1` which involves state changes to `O`. The serialisability property requires that a write lock must be acquired on `O` before it is modified; thus the body of `op1` should contain a call to the `setlock` operation of the concurrency controller:

```
public boolean op1 (...)
{
    if (setlock (new Lock(LockMode.WRITE) == LockResult.GRANTED)
    {
        // actual state change operations follow
        ...
    }
}
```

Program 1: Simple Concurrency Control

The operation `setlock`, provided by the `LockManager` class, performs the following functions in this case:

(i) Check write lock compatibility with the currently held locks, and if allowed:

(ii) Call the `StateManager` operation `activate` that will load, if not done already, the latest persistent state of `O` from the object store. Then call the `StateManager` operation `modified` which has the effect of creating an instance of either `RecoveryRecord` or `PersistenceRecord` for `O` depending upon whether `O` was persistent or not (the Lock is a `WRITE` lock so the old state of the object must be retained prior to modification) and inserting it into the `RecordList` of `Trans`.

(iii) Create and insert a `LockRecord` instance in the `RecordList` of `Trans`.

Now suppose that action `Trans` is aborted sometime after the lock has been acquired. Then the `rollback` operation of `AtomicAction` will process the `RecordList` instance associated with `Trans` by invoking an appropriate `Abort` operation on the various records. The implementation of this operation by the `LockRecord` class will release the `WRITE` lock while that of `RecoveryRecord/PersistenceRecord` will restore the prior state of `O`.

It is important to realise that all of the above work is automatically being performed by *JavaArjuna* and JavaArjuna protocol engine on behalf of the application programmer. The programmer need only start the transaction and set an appropriate lock; *JavaArjuna* takes care of Resource registration, persistence, concurrency control and recovery.

We shall now examine how each of the classes in the *JavaArjuna* hierarchy relates to JavaArjuna implementation described earlier, and the necessary persistence and concurrency control implementations.

2. Using JavaArjuna

In this section we shall describe *JavaArjuna* in more detail.

2.1 Configurable options

The following table shows the configuration features specific to JavaArjuna available in *JavaArjuna*, with default values shown in italics. For more detailed information, the relevant section numbers are provided. Those in shaded boxes are not available in the evaluation version of *JavaArjuna*.

Configuration Name	Possible Values	Relevant Section
OBJECTSTORE_SYNC	*ON*/OFF	Section 2.3.
OBJECTSTORE_TYPE	ShadowStore/*ShadowNoFileLockStore*	See Appendix
LOCKSTORE_TYPE	BasicLockStore/*BasicPersistentLockStore*	Section 2.5.1.
LOCKSTORE_DIR	Windows: *C:\JavaArjuna\LockStore* Unix: */usr/local/JavaArjuna/LockStore*	Section 2.5.1.

Table 1: JavaArjuna configuration options.

2.2 Object states

JavaArjuna needs to be able to remember the state of an object for several purposes, including recovery (the state represents some past state of the object), and for persistence (the state represents the final state of an object at application termination). Since all of these requirements require common functionality they are all implemented using the same mechanism - the classes `Input/OutputObjectState` and `Input/OutputBuffer`.

```
public class OutputBuffer
{
public OutputBuffer ();

public final synchronized boolean valid ();
public synchronized byte[] buffer();
public synchronized int length ();

      /* pack operations for standard Java types */
```

```
public synchronized void packByte (byte b) throws IOException;
public synchronized void packBytes (byte[] b) throws IOException;
public synchronized void packBoolean (boolean b) throws IOException;
public synchronized void packChar (char c) throws IOException;
public synchronized void packShort (short s) throws IOException;
public synchronized void packInt (int i) throws IOException;
public synchronized void packLong (long l) throws IOException;
public synchronized void packFloat (float f) throws IOException;
public synchronized void packDouble (double d) throws IOException;
public synchronized void packString (String s) throws IOException;
};

public class InputBuffer
{
public InputBuffer ();

public final synchronized boolean valid ();
public synchronized byte[] buffer();
public synchronized int length ();

      /* unpack operations for standard Java types */

public synchronized byte unpackByte () throws IOException;
public synchronized byte[] unpackBytes () throws IOException;
public synchronized boolean unpackBoolean () throws IOException;
public synchronized char unpackChar () throws IOException;
public synchronized short unpackShort () throws IOException;
public synchronized int unpackInt () throws IOException;
public synchronized long unpackLong () throws IOException;
public synchronized float unpackFloat () throws IOException;
public synchronized double unpackDouble () throws IOException;
public synchronized String unpackString () throws IOException;
};
```

The `Input/OutputBuffer` class maintains an internal array into which instances of the standard Java types can be contiguously packed (unpacked) using the `pack (unpack)` operations. This buffer is automatically resized as required should it have insufficient space. The instances are all stored in the buffer in a standard form (so-called network byte order) to make them machine independent.

```
class OutputObjectState extends OutputBuffer
{
public OutputObjectState (Uid newUid, String typeName);

public boolean notempty ();
public int size ();
public Uid stateUid ();
public String type ();
};
```

```
class InputObjectState extends InputBuffer
{
public OutputObjectState (Uid newUid, String typeName, byte[] b);

public boolean notempty ();
public int size ();
public Uid stateUid ();
public String type ();
};
```

The class `Input/OutputObjectState` provides all the functionality of `Input/OutputBuffer` (through inheritance) but adds two additional instance variables that signify the `Uid` and `type` of the object for which the `Input/OutputObjectState` instance is a compressed image. These are used when accessing the object store during storage and retrieval of the object state.

2.3 The object store

The object store provided with *JavaArjuna* deliberately has a fairly restricted interface so that it can be implemented in a variety of ways. For example, object stores are implemented in shared memory; on the Unix file system (in several different forms); and as a remotely accessible store. More complete information about the object stores available in *JavaArjuna* can be found in the *Appendix*. *Note*: as with all *JavaArjuna* classes the default object stores are pure Java implementations; to access the shared memory and other more complex object store implementations it is necessary to use native methods.

All of the object stores hold and retrieves instances of the class `Input/OutputObjectState`. These instances are named by the `Uid` and `Type` of the object that they represent. States are read using the `read_committed` operation and written by the system using the `write_uncommitted` operation. Under normal operation new object states do not overwrite old object states but are written to the store as *shadow copies*. These shadows replace the original only when the `commit_state` operation is invoked. Normally all interaction with the object store is performed by *JavaArjuna* system components as appropriate thus the existence of any shadow versions of objects in the store are hidden from the programmer.

```
public class ObjectStore
{
public static final int OS_COMMITTED;
public static final int OS_UNCOMMITTED;
public static final int OS_COMMITTED_HIDDEN;
public static final int OS_UNCOMMITTED_HIDDEN;
public static final int OS_UNKNOWN;

      /* The abstract interface */
public abstract boolean commit_state (Uid u, String name)
                                                throws ObjectStoreException;
public abstract InputObjectState read_committed (Uid u, String name)
                                            throws ObjectStoreException;
public abstract boolean write_uncommitted (Uid u, String name,
                        OutputObjectState os) throws ObjectStoreException;
      . . .
};
```

When a transactional object is committing it is necessary for it to make certain state changes persistent in order that it can recover in the event of a failure and either continue to commit, or rollback. When using JavaArjuna, *JavaArjuna* will take care of this automatically. To guarantee ACID properties, these state changes must be *flushed* to the persistence store implementation before the transaction can proceed to commit; if they are not, the application may assume that the transaction has committed when in fact the state changes may still reside within an operating system cache, and may be lost by a subsequent machine failure. By default, *JavaArjuna* ensures that such state changes are flushed. However, doing so can impose a significant performance penalty on the application. To prevent transactional object state flushes, set the `OBJECTSTORE_SYNC` variable to `OFF`.

2.3.1 Selecting an object store implementation

JavaArjuna comes with support for several different object store implementations. The *Appendix* describes these implementations, how to select and configure a given implementation (using the `OBJECTSTORE_TYPE` property variable) on a per object basis, and indicates how additional implementations can be provided.

2.4 StateManager

The *JavaArjuna* class `StateManager` manages the state of an object and provides all of the basic support mechanisms required by an object for state management purposes. `StateManager` is responsible for creating and registering appropriate `Resources` concerned with the persistence and recovery of the transactional object. If a transaction is nested, then `StateManager` will also propagate `Resource` information between child transactions and their parents at commit time.

Objects in *JavaArjuna* are assumed to be of three possible basic flavours. They may simply be *recoverable*, in which case `StateManager` will attempt to generate and maintain

appropriate recovery information for the object (as instances of the class `Input/OutputObjectState`) . Such objects have lifetimes that do not exceed the application program that creates them. Objects may be *recoverable and persistent*, in which case the lifetime of the object is assumed to be greater than that of the creating or accessing application so that in addition to maintaining recovery information `StateManager` will attempt to automatically load (unload) any existing persistent state for the object by calling the `activate` (`deactivate`) operation at appropriate times. Finally, objects may possess none of these capabilities in which case no recovery information is ever kept nor is object activation/deactivation ever automatically attempted. This object property is selected at object construction time and cannot be changed thereafter. Thus an object cannot gain (or lose) recovery capabilities at some arbitrary point during its lifetime.

```
public class ObjectStatus
{
public static final int PASSIVE;
public static final int PASSIVE_NEW;
public static final int ACTIVE;
public static final int ACTIVE_NEW;
public static final int UNKNOWN_STATUS;
};

public class ObjectType
{
public static final int RECOVERABLE;
public static final int ANDPERSISTENT;
public static final int NEITHER;
};

public abstract class StateManager
{
public synchronized boolean activate ();
public synchronized boolean activate (String storeRoot);
public synchronized boolean deactivate ();
public synchronized boolean deactivate (String storeRoot, boolean commit);

public synchronized void destroy ();

public final Uid get_uid ();

public boolean restore_state (InputObjectState, int ObjectType);
public boolean save_state (OutputObjectState, int ObjectType);
public String type ();
      . . .

protected StateManager ();
protected StateManager (int ObjectType, ObjectName attr);
protected StateManager (Uid uid);
protected StateManager (Uid uid, ObjectName attr);
      . . .

protected final void modified ();
      . . .
};


public class ObjectModel
{
public static final int SINGLE;
public static final int MULTIPLE;
};
```

If an object is recoverable (or persistent) then `StateManager` will invoke the operations `save_state` (while performing `deactivation`), `restore_state` (while performing `activate`) and `type` at various points during the execution of the application. These operations *must* be implemented by the programmer since `StateManager` does not have access to a runtime

description of the layout of an arbitrary Java object in memory and thus cannot implement a default policy for converting the in memory version of the object to its passive form. However, the capabilities provided by `Input/OutputObjectState` make the writing of these routines fairly simple. For example, the `save_state` implementation for a class `Example` that had member variables called `A`, `B` and `C` could simply be the following:

```
public boolean save_state ( OutputObjectState os, int ObjectType )
{
    if (!super.save_state(os, ObjectType))
        return false;

    try
    {
       os.packInt(A);
       os.packString(B);
       os.packFloat(C);

       return true;
    }
    catch (IOException e)
    {
       return false;
    }
}
```

In order to support crash recovery for persistent objects it is necessary for all `save_state` and `restore_state` methods of user objects to call `super.save_state` and `super.restore_state`.

The `get_uid` operation of `StateManager` provides read only access to an object's internal system name for whatever purpose the programmer requires (such as registration of the name in a name server). The value of the internal system name can *only* be set when an object is initially constructed - either by the provision of an explicit parameter or by generating a new identifier when the object is created.

The `destroy` method can be used to remove the object's state from the object store. This is an atomic operation, and therefore will only remove the state if the top-level transaction within which it is invoked eventually commits. The programmer *must* obtain exclusive access to the object prior to invoking this operation.

Since object recovery and persistence essentially have complimentary requirements (the only difference being where state information is stored and for what purpose) `StateManager` effectively combines the management of these two properties into a single mechanism. That is, it uses instances of the class `Input/OutputObjectState` both for recovery and persistence purposes. An additional argument passed to the `save_state` and `restore_state` operations allows the programmer to determine the purpose for which any given invocation is being made thus allowing different information to be saved for recovery and persistence purposes.

2.4.1 Object models

JavaArjuna supports two models for objects, which as we shall show affect how an objects state and concurrency control are implemented:

- SINGLE: only a single copy of the object exists within the application; this will reside within a single server, and all clients must address their invocations to this server. This model provides better performance, but represents a single point of failure, and in a multi-threaded environment may not protect the object from corruption if a single thread fails.

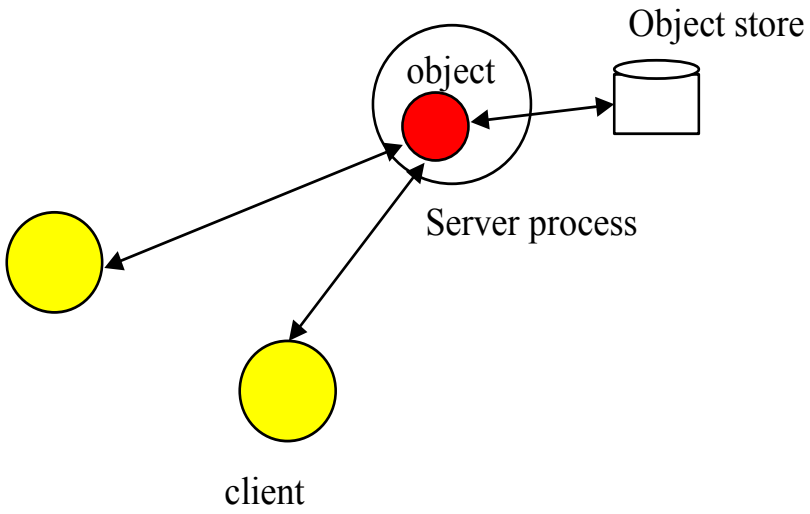

Figure 4: SINGLE object model.

- MULTIPLE: logically a single instance of the object exists, but copies of it are distributed across different servers; the performance of this model is worse than the SINGLE model, but it provides better failure isolation.

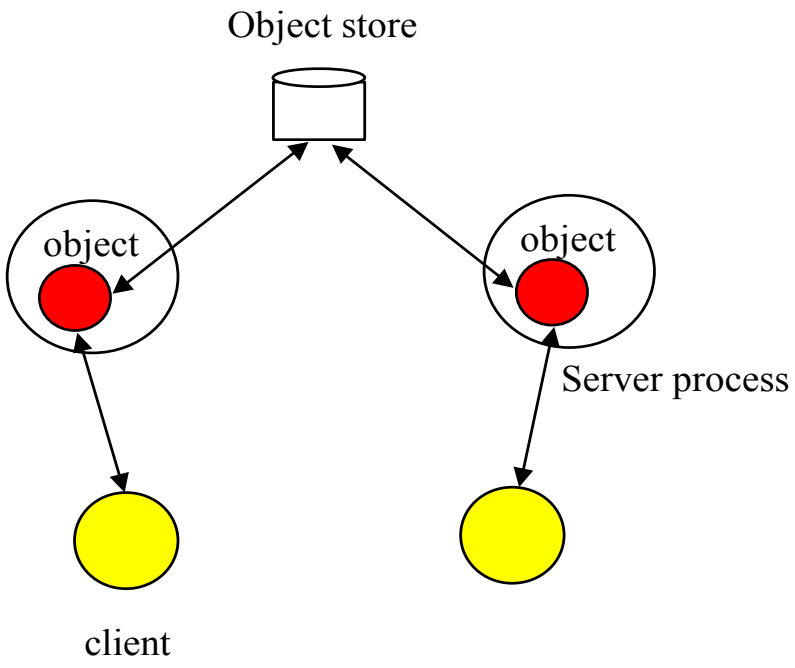

Figure 5: MULTIPLE object model.

The default model is SINGLE. The programmer can override this on a per object basis by providing an appropriate instance of the `com.arjuna.ArjunaCommon.ClassLib/StateManagerAttribute` class at object construction. Note, the model can be changed between each successive instantiation of the object, i.e., it need not be the same during the object's lifetime.

2.4.2 Summary

In summary, the *JavaArjuna* class `StateManager` manages the state of an object and provides all of the basic support mechanisms required by an object for state management purposes. Some operations *must* be defined by the class developer. These operations are: `save_state`, `restore_state`, and `type`.

```
boolean save_state (OutputObjectState state, int ObjectType)
```

Invoked whenever the state of an object might need to be saved for future use - primarily for recovery or persistence purposes. The `ObjectType` parameter indicates the reason that `save_state` was invoked by *JavaArjuna*. This enables the programmer to save different pieces of information into the `OutputObjectState` supplied as the first parameter depending upon whether the state is needed for recovery or persistence purposes. For example, pointers to other *JavaArjuna* objects might be saved simply as pointers for recovery purposes but as Uid's for persistence purposes. As shown earlier, the `OutputObjectState` class provides convenient operations to allow the saving of instances of all of the basic types in Java. In order to support crash recovery for persistent objects it is necessary for all `save_state` methods to call `super.save_state`.

Note: `save_state` assumes that an object is *internally consistent* and that all variables saved have valid values. *It is the programmer's responsibility to ensure that this is the case.*

```
boolean restore_state (InputObjectState state, int ObjectType)
```

Invoked whenever the state of an object needs to be restored to the one supplied. Once again the second parameter allows different interpretations of the supplied state. In order to support crash recovery for persistent objects it is necessary for all `restore_state` methods to call `super.restore_state`.

```
String type ()
```

The *JavaArjuna* persistence and concurrency control mechanisms *require* a means of determining the type of an object as a string. By convention this information indicates the position of the class in the hierarchy. For example, `"/StateManager/LockManager/Object"`[1].

2.4.3 Example

Consider the following basic `Array` class derived from the `StateManager` class (in this example, to illustrate saving and restoring of an object's state, the `highestIndex` variable is used to keep track of the highest element of the array that has a non-zero value):

```
public class Array extends StateManager
{
public Array ();
public Array (Uid objUid);
public void finalize ( super.terminate(); };

      /* Class specific operations. */

public boolean set (int index, int value);
public int get (int index);

      /* State management specific operations. */

public boolean save_state (OutputObjectState os, int ObjectType);
public boolean restore_state (InputObjectState os, int ObjectType);
public String type ();

public static final int ARRAY_SIZE = 10;

private int[] elements = new int[ARRAY_SIZE];
private int highestIndex;
};
```

The `save_state`, `restore_state` and `type` operations can be defined as follows:

[1] In Java this method is not strictly necessary, but is provided for interface compatibility.

```
/* Ignore ObjectType parameter for simplicity */

public boolean save_state (OutputObjectState os, int ObjectType)
{
    if (!super.save_state(os, ObjectType))
        return false;

    try
    {
      packInt(highestIndex);

      /*
       * Traverse array state that we wish to save. Only save active elements
       */

      for (int i = 0; i <= highestIndex; i++)
            os.packInt(elements[i]);

       return true;
    }
    catch (IOException e)
    {
        return false;
    }
}

public boolean restore_state (InputObjectState os, int ObjectType)
{
    if (!super.restore_state(os, ObjectType))
        return false;

    try
    {
      int i = 0;

      highestIndex = os.unpackInt();

      while (i < ARRAY_SIZE)
      {
            if (i <= highestIndex)
                  elements[i] =  os.unpackInt();
            else
                  elements[i] = 0;
            i++;
      }

      return true;
    }
    catch (IOException e)
    {
       return false;
    }
}
```

```
public String type ()
{
      return "/StateManager/Array";
}
```

2.5 The lock store

Concurrency control information within *JavaArjuna* is maintained by *locks*. Locks which are required to be shared between objects in different processes may be held within a lock store, similar to the object store facility presented previously. The lock store provided with *JavaArjuna* deliberately has a fairly restricted interface so that it can be implemented in a variety of ways. For example, lock stores are implemented in shared memory; on the Unix file system (in several different forms); and as a remotely accessible store. More information about the object stores available in *JavaArjuna* can be found in the *Appendix*.

Note: as with all *JavaArjuna* classes the default lock stores are pure Java implementations; to access the shared memory and other more complex lock store implementations it is necessary to use native methods.

```
public class LockStore
{
public abstract InputObjectState read_state (Uid u, String tName)
                                                    throws
LockStoreException;

public abstract boolean remove_state (Uid u, String tname);
public abstract boolean write_committed (Uid u, String tName,
                                         OutputObjectState state);
};
```

2.5.1 Selecting a lock store implementation

JavaArjuna comes with support for several different object store implementations. If the object model being used is SINGLE, then no lock store is required for maintaining locks, since the information about the object is not exported from it. However, if the MULTIPLE model is used, then different run-time environments (processes, Java virtual machines) may need to share concurrency control information. The implementation type of the lock store to use can be specified for all objects within a given execution environment using the LOCKSTORE_TYPE property variable. Currently this can have one of the following values:

- *BasicLockStore*: this is an in-memory implementation which does not, by default, allow sharing of stored information between execution environments. The application programmer is responsible for sharing the store information.
- *BasicPersistentLockStore*: this is the default implementation, and stores locking information within the local file system. Therefore execution environments *that share the same file store* can share concurrency control information. The root of the file system into which locking

information is written is the `LockStore` directory within the *JavaArjuna* installation directory. This can be overridden at runtime by setting the LOCKSTORE_DIR property variable accordingly, or placing the location within the CLASSPATH:

```
java -DLOCKSTORE_DIR=/var/tmp/LockStore myprogram

or

java –classpath $CLASSPATH;/var/tmp/LockStore myprogram
```

If neither of these approaches is taken, then the default location will be at the same level as the `etc` directory of the installation.

2.6 LockManager

The concurrency controller is implemented by the class `LockManager` which provides sensible default behaviour while allowing the programmer to override it if deemed necessary by the particular semantics of the class being programmed. The primary programmer interface to the concurrency controller is via the `setlock` operation. By default, the *JavaArjuna* runtime system enforces strict two-phase locking following a multiple reader, single writer policy on a per object basis. Lock acquisition is under programmer control, since just as `StateManager` cannot determine if an operation modifies an object, `LockManager` cannot determine if an operation requires a read or write lock. Lock release, however, is normally under control of the system and requires no further intervention by the programmer. This ensures that the two-phase property can be correctly maintained.

The `LockManager` class is primarily responsible for managing requests to set a lock on an object or to release a lock as appropriate. However, since it is derived from `StateManager`, it can also control when some of the inherited facilities are invoked. For example, if a request to set a write lock is granted, then `LockManager` invokes `modified` directly assuming that the setting of a write lock implies that the invoking operation must be about to modify the object. This may in turn cause recovery information to be saved if the object is recoverable. In a similar fashion, successful lock acquisition causes `activate` to be invoked.

Therefore, in terms of JavaArjuna, `LockManager` is directly responsible for activating/de-activating persistent objects, and registering `Resources` for managing concurrency control. By driving the `StateManager` class, it is also responsible for registering `Resources` for persistent/recoverable state manipulation and object recovery. The application programmer simply sets appropriate locks, starts and ends transactions, and extends the `save_state` and `restore_state` methods of `StateManager`.

```
public class LockResult
{
public static final int GRANTED;
public static final int REFUSED;
public static final int RELEASED;
};

public class ConflictType
{
public static final int CONFLICT;
public static final int COMPATIBLE;
public static final int PRESENT;
};

public abstract class LockManager extends StateManager
{
public static final int defaultTimeout;
public static final int defaultRetry;
public static final int waitTotalTimeout;

public synchronized int setlock (Lock l);
public synchronized int setlock (Lock l, int retry);
public synchronized int setlock (Lock l, int retry, int sleepTime);
public synchronized boolean releaselock (Uid uid);

      /* abstract methods inherited from StateManager */

public boolean restore_state (InputObjectState os, int ObjectType);
public boolean save_state (OutputObjectState os, int ObjectType);
public String type ();

protected LockManager ();
protected LockManager (int ObjectType, ObjectName attr);
protected LockManager (Uid storeUid);
protected LockManager (Uid storeUid, int ObjectType, ObjectName attr);
      . . .
};
```

The `setlock` operation must be parameterised with the type of lock required (`READ` / `WRITE`), and the number of retries to acquire the lock before giving up. If a lock conflict occurs, one of the following scenarios will take place:

1) If the retry value is equal to `LockManager.waitTotalTimeout`, then the thread which called `setlock` will be blocked until the lock is released, or the total timeout specified has elapsed, and in which `REFUSED` will be returned.

2) If the lock cannot be obtained initially then `LockManager` will try for the specified number of retries, waiting for the specified timeout value between each failed attempt. The default is 100 attempts, each attempt being separated by a 0.25 seconds delay; the time between retries is specified in *micro-seconds*.

If a lock conflict occurs the current implementation simply times out lock requests, thereby preventing deadlocks, rather than providing a full deadlock detection scheme. If the requested lock is obtained, the `setlock` operation will return the value `GRANTED`, otherwise the value `REFUSED` is returned. It is the responsibility of the programmer to ensure that the remainder of the code for an operation is only executed if a lock request is granted. Below are examples of the use of the `setlock` operation.

```
res = setlock(new Lock(WRITE), 10); // Will attempt to set a
                                    // write lock 11 times (10
                                    // retries) on the object
                                    // before giving up.

res = setlock(new Lock(READ), 0);         // Will attempt to set a read
                                    // lock 1 time (no retries) on
                                    // the object before giving up.

res = setlock(new Lock(WRITE);            // Will attempt to set a write
                                    // lock 101 times (default of
                                    // 100 retries) on the object
                                    // before giving up.
```

The concurrency control mechanism is integrated into the atomic action mechanism, thus ensuring that as locks are granted on an object appropriate information is registered with the currently running atomic action to ensure that the locks are released at the correct time. This frees the programmer from the burden of explicitly freeing any acquired locks if they were acquired within atomic actions. However, if locks are acquired on an object *outside* of the scope of an atomic action, it is the programmer's responsibility to release the locks when required, using the corresponding `releaselock` operation.

2.6.1 Locking policy

Unlike many other systems, locks in *JavaArjuna* are not special system types. Instead they are simply instances of other *JavaArjuna* objects (the class `Lock` which is also derived from `StateManager` so that locks may be made persistent if required and can also be named in a simple fashion). Furthermore, `LockManager` deliberately has no knowledge of the semantics of the actual policy by which lock requests are granted. Such information is maintained by the actual `Lock` class instances which provide operations (the `conflictsWith` operation) by which `LockManager` can determine if two locks conflict or not. This separation is important in that it allows the programmer to derive new lock types from the basic `Lock` class and by providing appropriate definitions of the conflict operations enhanced levels of concurrency may be possible.

```
public class LockMode
{
public static final int READ;
public static final int WRITE;
};

public class LockStatus
{
public static final int LOCKFREE;
public static final int LOCKHELD;
public static final int LOCKRETAINED;
};

public class Lock extends StateManager
{
public Lock (int lockMode);

public boolean conflictsWith  (Lock otherLock);
public boolean modifiesObject ();

public boolean restore_state (InputObjectState os, int ObjectType);
public boolean save_state (OutputObjectState os, int ObjectType);
public String type ();
      . . .
 };
```

The `Lock` class provides a `modifiesObject` operation which `LockManager` uses to determine if granting this locking request requires a call on `modified`. This operation is provided so that locking modes other than simple read and write can be supported. The supplied `Lock` class supports the traditional multiple reader/single writer policy.

2.6.2 Object construction and destruction

Recall that *JavaArjuna* objects can be recoverable; recoverable and persistent; or neither. Additionally each object possesses a unique internal name. These attributes can *only* be set when that object is constructed. Thus `LockManager` provides two *protected* constructors for use by derived classes, each of which fulfils a distinct purpose:

```
LockManager ():
```

This constructor allows the creation of *new* objects, that is, no prior state is assumed to exist.

```
LockManager (int ObjectType, ObjectName attr):
```

As above, this constructor allows the creation of *new* objects, that is, no prior state is assumed to exist. The `ObjectType` parameter determines whether an object is simply recoverable (indicated by `RECOVERABLE`); recoverable and persistent (indicated by `ANDPERSISTENT`) or neither (`NEITHER`). If an object is marked as being persistent then the state of the object will be stored in one of the object stores. The `shared` parameter only

has meaning if `ot` is `RECOVERABLE`; if `attr` is not null and the object model is `SINGLE` (the default behaviour) then the recoverable state of the object is maintained within the object itself (i.e., it has no external representation), otherwise an in-memory (volatile) object store is used to store the state of the object between atomic actions.

Constructors for new persistent objects should make use of atomic actions within themselves. This will ensure that the state of the object is automatically written to the object store either when the action in the constructor commits or, if an enclosing action exists, when the appropriate top-level action commits. Later examples in this chapter illustrate this point further.

```
LockManager(Uid objUid):
```

This constructor allows access to an *existing* persistent object, whose internal name is given by the `objUid` parameter. Objects constructed using this operation will normally have their prior state (identified by `objUid`) loaded from an object store automatically by the system.

```
LockManager(Uid objUid, ObjectName attr):
```

As above, this constructor allows access to an *existing* persistent object, whose internal name is given by the `objUid` parameter. Objects constructed using this operation will normally have their prior state (identified by `objUid`) loaded from an object store automatically by the system. If the `attr` parameter is not null, and the object model is `SINGLE` (the default behaviour), then the object will not be reactivated at the start of each top-level transaction.

The destructor of a programmer-defined class *must* invoke the inherited operation `terminate` to inform the state management mechanism that the object is about to be destroyed otherwise unpredictable results may occur.

Because `LockManager` inherits from `StateManager`, it will pass any supplied `ObjectName` instance to the `StateManager` class. As such, it is possible to set the StateManager object model as described in Section 2.4.1.

2.7 Transaction issues

Atomic actions can be used by both applications programmers and class developers. Thus entire operations (or parts of operations) can be made atomic as required by the semantics of a particular operation.

2.7.1 Checking transactions

In a multi-threaded application, multiple threads may be associated with a transaction during its lifetime, i.e., the thread's share the context. In addition, it is possible that if one thread terminates a transaction other threads may still be active within it. In a distributed environment,

it can be difficult to guarantee that all threads have finished with a transaction when it is terminated. By default, *JavaArjuna* will issue a warning if a thread terminates a transaction when other threads are still active within it; however, it will allow the transaction termination to continue. Other solutions to this problem are possible, e.g., blocking the thread which is terminating the transaction until all other threads have disassociated themselves from the transaction context. Therefore, *JavaArjuna* provides the `CheckedAction` class, which allows the thread/transaction termination policy to be overridden. Each transaction has an instance of this class associated with it, and application programmers can provide their own implementations on a per transaction basis.

```
public class CheckedAction
{
public CheckedAction ();

public synchronized void check (boolean isCommit, Uid actUid,
                                BasicList list);
};
```

When a thread attempts to terminate the transaction and there are active threads within it, the system will invoke the `check` method on the transaction's `CheckedAction` object. The parameters to the `check` method are:

- `isCommit`: indicates whether the transaction is in the process of committing or rolling back.
- `actUid`: the transaction identifier.
- `list`: a list of all of the threads currently marked as active within this transaction.

When `check` returns, the transaction termination will continue. Obviously the state of the transaction at this point may be different from that when `check` was called, e.g., the transaction may subsequently have been committed.

2.7.2 Nested transactions

There are no special constructs for nesting of transactions: if an action is begun while another action is running then it is automatically nested. This allows for a modular structure to applications, whereby objects can be implemented using atomic actions within their operations without the application programmer having to worry about the applications which use them, i.e., whether or not the applications will use atomic actions as well. Thus, in some applications actions may be top-level, whereas in others they may be nested. Objects written in this way can then be shared between application programmers, and *JavaArjuna* will guarantee their consistency.

If a nested action is aborted then all of its work will be undone, although strict two-phase locking means that any locks it may have obtained will be retained until the top-level action commits or aborts. If a nested action commits then the work it has performed will only be

committed by the system if the top-level action commits; if the top-level action aborts then *all* of the work will be undone.

The committing or aborting of a nested action does not automatically affect the outcome of the action within which it is nested. This is application dependant, and allows a programmer to structure atomic actions to contain faults, undo work, etc.

2.7.3 Independent top-level transactions

In addition to normal top-level and nested atomic actions *JavaArjuna* also supports *independent top-level* actions, which can be used to relax strict serialisability in a controlled manner. An independent top-level action can be executed from anywhere within another atomic action and behaves *exactly* like a normal top-level action, that is, its results are made permanent when it commits and will not be undone if any of the actions within which it was originally nested abort.

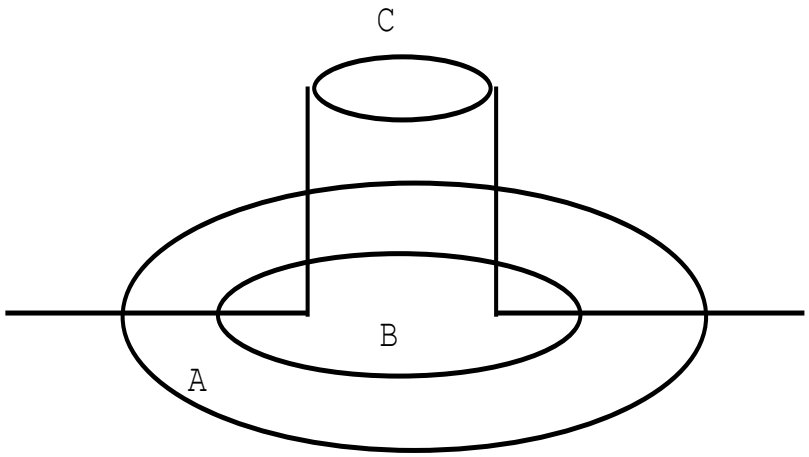

Figure 6: Independent Top-Level Action

Figure 6 shows a typical nesting of atomic actions, where action `B` is nested within action `A`. Although atomic action `C` is logically nested within action `B` (it had its `Begin` operation invoked while `B` was active) because it is an independent top-level action, it will commit or abort independently of the other actions within the structure. Because of the nature of independent top-level actions they should be used with caution and only in situations where their use has been carefully examined.

Top-level actions can be used within an application by declaring and using instances of the class `TopLevelAction`. They are used in exactly the same way as other transactions.

2.7.4 Transactions within `save_state` and `restore_state`

Caution must be exercised when writing the `save_state` and `restore_state` operations to ensure that no atomic actions are started (either explicitly in the operation or implicitly through use of some other operation). This restriction arises due to the fact that *JavaArjuna* may

invoke `restore_state` as part of its commit processing resulting in the attempt to execute an atomic action during the commit or abort phase of another action. This might violate the atomicity properties of the action being committed (aborted) and is thus discouraged.

2.8 Example

If we consider the `Array` example given previously, the `set` and `get` operations could be implemented as shown below. *Note*: this is a simplification of the code, ignoring error conditions and exceptions.

```
public boolean set (int index, int value)
{
      boolean result = false;
      AtomicAction A = new AtomicAction();

      A.begin();

      // We need to set a WRITE lock as we want to modify the state.

      if (setlock(new Lock(LockMode.WRITE), 0) == LockResult.GRANTED)
      {
            elements[index] = value;
            if ((value > 0) &&(index > highestIndex
                  highestIndex = index;
            A.commit(true);
            result = true;
      }
      else
            A.rollback();

      return result;
}

public int get (int index)  // assume -1 means error
{
      AtomicAction A = new AtomicAction();

      A.begin();

      // We only need a READ lock as the state is unchanged.

      if (setlock(new Lock(LockMode.READ), 0) == LockResult.GRANTED)
      {
            A.commit(true);

             return elements[index];
      }
      else
            A.rollback();

      return -1;
}
```

2.9 Garbage collecting objects

Java objects are deleted when the garbage collector determines that they are no longer required. Deleting an object that is currently under the control of a transaction *must* be approached with caution since if the object is being manipulated within a transaction its fate is effectively determined by the transaction. Therefore, regardless of the references to a transactional object maintained by an application, *JavaArjuna* will always retain its own

references to ensure that the object is not garbage collected until after any transaction has terminated.

3. Constructing a *JavaArjuna* application

There are two distinct phases to the development of a *JavaArjuna* application:

(i) Developing new classes with certain characteristics (for example, Persistent, Recoverable, Concurrency Controlled).

(ii) Developing the application(s) that make use of the new classes of objects.

Although these two phases may be performed in parallel and by a single person, we shall refer to the first step as the job of the *class developer* and the second as the job of the *applications developer*. The class developer will be concerned about defining appropriate `save_state` and `restore_state` operations for the class, setting appropriate locks in operations, and invoking the appropriate *JavaArjuna* class constructors. The applications developer will be more concerned with defining the general structure of the application, particularly with regards to the use of atomic actions.

This chapter illustrates the points made in previous sections by outlining a simple application - in this case a simple FIFO `Queue` class for integer values will be developed. The implementation of the `Queue` will be with a doubly linked list structure, and it will be implemented as a single object. We shall be using this example throughout the rest of this manual to help illustrate the various mechanisms provided by *JavaArjuna*. While this is an unrealistic example application it enables all of the *JavaArjuna* modifications to be described without requiring in depth knowledge of the application code. The code for all versions of this application is available in the *JavaArjuna* software release in the directory `Examples/Queue`.

3.1 Queue description

The queue is a traditional FIFO queue, where elements are added to the front and removed from the back. The operations provided by the queue class allow the values to be placed on to the queue (`enqueue`) and to be removed from it (`dequeue`), and it is also possible to change or inspect the values of elements in the queue.

In this example implementation, an array is used to represent the queue. A limit of `QUEUE_SIZE` elements has been imposed for this example.

The Java interface definition of this simple queue class is given below:

```
public class TransactionalQueue extends LockManager
                                implements TIE_CLASS_(Queue)
{
public TransactionalQueue (Uid uid);
public TransactionalQueue ();
public void finalize ();

public void enqueue (int v, Control c) throws OverFlow, UnderFlow,
                                              QueueError, Conflict;
public void dequeue  (IntHolder v, Control c) throws OverFlow, UnderFlow,
                                                     QueueError, Conflict;

public int queueSize (Control c);
public void inspectValue (int i, IntHolder v, Control c) throws OverFlow,
                                             UnderFlow, QueueError, Conflict;
public void setValue (int i, IntHolder v, Control c) throws OverFlow,
                                             UnderFlow, QueueError, Conflict;

public boolean save_state (OutputObjectState os, int ObjectType);
public boolean restore_state (InputObjectState os, int ObjectType);
public String type ();

public static final int QUEUE_SIZE = 40; // maximum size of the queue

private int[QUEUE_SIZE] elements;
private int numberOfElements;
};
```

3.2 Constructors and destructors

As stated in the previous section, to use an existing persistent object requires the use of a special constructor that is required to take the `Uid` of the persistent object; the implementation of such a constructor is given below:

```
public TransactionalQueue (Uid u)
{
    super(u);

    numberOfElements = 0;
}
```

The constructor that creates a new persistent object is similar:

```
public TransactionalQueue ()
{
    super(ObjectType.ANDPERSISTENT);

    numberOfElements = 0;

    try
    {
      AtomicAction A = new AtomicAction();

      A.begin(0); // Try to start atomic action

      // Try to set lock

      if (setlock(new Lock(LockMode.WRITE), 0) == LockResult.GRANTED)
      {
            A.commit(true);   // Commit
      }
      else                    // Lock refused so abort the atomic action
            A.rollback();
    }
    catch (Exception e)
    {
        System.err.println("Object construction error: "+e);
        System.exit(1);
    }
}
```

The use of an atomic action within the constructor for a new object follows the guidelines outlined earlier and ensures that the object's state will be written to the object store when the appropriate top level atomic action commits (which will either be the action `A` or some enclosing action active when the `TransactionalQueue` was constructed - since we are using explicit propagation in the example, the constructor would have to be modified to take a `Control`). The use of atomic actions in a constructor is simple: an action must first be declared and its b`egin` operation invoked; the operation must then set an appropriate lock on the object (in this case a `WRITE` lock must be acquired), then the main body of the constructor is executed. If this is successful the atomic action can be committed, otherwise it is aborted.

The destructor of the queue class is only required to call the `terminate` operation of `LockManager`

```
public void finalize ()
{
      super.terminate();
}
```

3.3 save_state, restore_state and type

The implementations of `save_state` and `restore_state` are relatively simple for this example:

```
public boolean save_state (OutputObjectState os, int ObjectType)
{
    if (!super.save_state(os, ObjectType))
        return false;

    try
    {
        os.packInt(numberOfElements);

        if (numberOfElements > 0)
        {
            for (int i = 0; i < numberOfElements; i++)
               os.packInt(elements[i]);
        }

        return true;
    }
    catch (IOException e)
    {
        return false;
    }
}

public boolean restore_state (InputObjectState os, int ObjectType)
{
    if (!super.restore_state(os, ObjectType))
        return false;

    try
    {
        numberOfElements = os.unpackInt();

        if (numberOfElements > 0)
        {
            for (int i = 0; i < numberOfElements; i++)
                elements[i] = os.unpackInt();
        }

        return true;
    }
    catch (IOException e)
    {
        return false;
    }
}
```

Because the `Queue` class is derived from the `LockManager` class, the operation `type` should be:

```
public String type ()
{
      return "/StateManager/LockManager/TransactionalQueue";
}
```

3.4 enqueue/dequeue operations

If the operations of the queue class are to be coded as atomic actions, then the `enqueue` operation could have the structure given below (the `dequeue` operation would be similarly structured):

```
public void enqueue (int v, Control ctx) throws OverFlow, UnderFlow,
QueueError
{
    AtomicAction A = new AtomicAction();
    boolean res = false;

      A.begin(0);

      if (setlock(new Lock(LockMode.WRITE), 0) == LockResult.GRANTED)
      {
            if (numberOfElements < QUEUE_SIZE)
            {
                  elements[numberOfElements] = v;
                    numberOfElements++;
                    res = true;
            }
            else
             {
                    A.rollback();
                  throw new UnderFlow();
             }
      }

      if (res)
            A.commit(true);
      else
      {
            A.rollback();
             throw new Conflict();
      }
}
```

3.5 queueSize

The implementation of `queueSize` is shown below:

```
public int queueSize (Control ctx) throws QueueError, Conflict
{
    AtomicAction A = new AtomicAction();
    int size = -1;
```

```
    try
    {
        A.begin(0);

        if (setlock(new Lock(LockMode.READ), 0) == LockResult.GRANTED)
            size = numberOfElements;

        if (size != -1)
          A.commit(true);
        else
        {
           A.rollback();

           throw new Conflict();
        }
    }
    catch (Exception e1)
    {
    }


    return size;
}
```

3.6 inspectValue/setValue operations

The implementation of `inspectValue` is shown below. `setValue` is similar, and not shown.

```
public void inspectValue (int index, IntHolder v,
                          Control ctx) throws UnderFlow,
                                              OverFlow, Conflict, QueueError
{
    AtomicAction A = new AtomicAction();
    boolean res = false;

    try
    {
        A.begin();

        if (setlock(new Lock(LockMode.READ), 0) == LockResult.GRANTED)
        {
            if (index < 0)
            {
                A.rollback();
                throw new UnderFlow();
            }
            else
            {
                // array is 0 - numberOfElements -1
```

```
                if (index > numberOfElements -1)
                {
                    A.rollback();
                    throw new OverFlow();
                }
                else
                {
                    v.value = elements[index];
                    res = true;
                }
            }
        }

        if (res)
            A.commit(true);
        else
        {
            A.rollback();
            throw new Conflict();
        }
    }
    catch (Exception e1)
    {
    }
}
```

3.7 The client

Rather than show all of the code for the client, we shall concentrate on a representative portion. Before invoking one of the queue's operations involves, the client starts a transaction and obtains its context (`Control`). This is then passed as an extra parameter to each method. If the client does not start a transaction, it can pass nil as the parameter; each of the object's methods will then be invoked within a top-level transaction. The `queueSize` operation is shown below:

```
AtomicAction A = new AtomicAction();
int size = 0;
Control ctx = null;

try
{
    A.begin(0);

    try
    {
        ctx = A.control();
    }
    catch (Exception e)
    {
        System.err.println("Error - could not get transaction control.");
        size = -1;
```

```
        ctx = null;
    }

    try
    {
        if (ctx != null)
            size = queue.queueSize(ctx);
    }
    catch (Exception e)
    {
    }

    if (size >= 0)
    {
        A.commit(true);

        System.out.println("Size of queue: "+size);
    }
    else
        A.rollback();
}
catch (Exception e)
{
    System.err.println("Caught unexpected exception!");
}

ctx = null;
```

3.8 Comments

Since the queue object is persistent, then the state of the object will survive any failures of the node on which it is located. The state of the object that will survive is that produced by the last top-level committed atomic action performed on the object. If it is the intention of an application to perform two `enqueue` operations atomically, for example, then this can be done by nesting the `enqueue` operations in another enclosing atomic action. In addition, concurrent operations on such a persistent object will be serialised, thereby preventing inconsistencies in the state of the object. However, since the elements of the queue objects are not individually concurrency controlled, certain combinations of concurrent operation invocations will be executed serially, whereas logically they could be executed concurrently. For example, modifying the states of two different elements in the queue. In the next section we address some of these issues.

4. Useful hints and tips

4.1 General

4.1.1 Memory management

Memory management is typically a difficult issue when applications are distributed. JavaArjuna has no way of allowing an application to specify that it is no longer interested in a particular transaction. A transaction has no way of knowing how many references to it are remaining in the distributed system. Thus, there is no clean way of knowing when to remove a transaction; simply leaving transactions around forever results a build up of garbage. Therefore, *JavaArjuna* takes the following memory management decisions for transactions:

- transactions invoked through the `Current` interface are removed from the system when they terminate, i.e., are committed or rolled back. Invocations on references to these transactions (`Coordinator` and `Terminator` references) after termination will result in exceptions which the programmer must catch.
- transactions created through a `TransactionFactory` are lazily removed from the system once they have terminated. There is no guarantee on the lifetime of a terminated transaction implementation.

In a future release of *JavaArjuna* it will be possible for programmers to override these defaults and have more control over the management of transactions.

Because Java relies upon the garbage collector to remove unused objects, it is possible that an application may run out of memory. If this happens then *JavaArjuna* will attempt to catch any `OutOfMemoryErrors` which are thrown and run the garbage collector before throwing a `NO_MEMORY` standard exception.

4.1.2 Using transactions in constructors

Examples throughout this manual have used transactions in the implementation of constructors for *new* persistent objects. This is deliberate because it guarantees correct propagation of the state of the object to the object store. Recall that the state of a modified persistent object is only written to the object store when the top-level transaction commits. Thus, if the constructor transaction is top-level and it commits, then the newly created object is written to the store and becomes available immediately. If however, the constructor transaction commits but is nested because some other transaction started prior to object creation is running, then the state will be written only if all of the parent transactions commit.

On the other hand, if the constructor does not use transactions then it is possible for inconsistencies in the system to arise. For example, if no transaction is active when the object is

created then its state will not be saved to the store until the next time the object is modified under the control of some transaction.

Consider this simple example:

```
AtomicAction A = new AtomicAction();
Object obj1;
Object obj2;

obj1 = new Object();                // create new object
obj2 = new Object("old");           // existing object

A.begin(0);
obj2.remember(obj1.get_uid());      // obj2 now contains reference to obj1
A.commit(true);                     // obj2 saved but obj1 is not
```

Here the two objects are created outside of the control of the top-level action `A`. `obj1` is a new object; `obj2` an old existing object. When the `remember` operation of `obj2` is invoked the object will be activated and the `Uid` of `obj1` remembered. Since this action commits the persistent state of `obj2` could now contain the `Uid` of `obj1`. However, the state of `obj1` itself has not been saved since it has not been manipulated under the control of any action. In fact, unless it is modified under the control of some action later in the application it will *never* be saved. If, however, the constructor had used an atomic action the state of `obj1` would have automatically been saved at the time it was constructed and this inconsistency could not arise.

4.1.3 More on save_state and restore_state

JavaArjuna may invoke the user-defined `save_state` operation of an object effectively at any time during the lifetime of an object including during the execution of the *body* of the object's constructor (particularly if it uses atomic actions). It is important, therefore, that all of the variables saved by `save_state` are correctly initialised.

Caution must be also exercised when writing the `save_state` and `restore_state` operations to ensure that no transactions are started (either explicitly in the operation or implicitly through use of some other operation). This restriction arises due to the fact that *JavaArjuna* may invoke `restore_state` as part of its commit processing resulting in the attempt to execute an atomic transaction during the commit or abort phase of another transaction. This might violate the atomicity properties of the transaction being committed (aborted) and is thus discouraged.

In order to support crash recovery for persistent objects it is necessary for all `save_state` and `restore_state` methods of user objects to call `super.save_state` and `super.restore_state`.

4.1.4 Packing objects

All of the basic types of Java (int, long, etc.) can be saved and restored from an `Input/OutputObjectState` instance by using the `pack` (and `unpack`) routines provided by

`Input/OutputObjectState`. However packing and unpacking objects should be handled differently. This is because packing objects brings in the additional problems of *aliasing*. That is two different object references may in actual fact point at the same item. For example:

```
public class Test
{
public Test (String s);
       ...
private String s1;
private String s2;
};

public Test (String s)
{
       s1 = s;
       s2 = s;
}
```

Here, both `s1` and `s2` point at the same string and a naive implementation of `save_state` could end up by copying the string twice. From a `save_state` perspective this is simply inefficient. However, it makes `restore_state` incorrect since it would unpack the two strings into different areas of memory destroying the original aliasing information. The current version of *JavaArjuna* will pack and unpack separate object references.

4.2 Direct use of StateManager

The examples throughout this manual have always derived user classes from `LockManager`. The reasons for this are twofold. Firstly, and most importantly, the serialisability constraints of atomic actions require it, and secondly it reduces the need for programmer intervention. However, if only access to *JavaArjuna's* persistence and recovery mechanisms is required, direct derivation of a user class from `StateManager` is possible.

Classes derived directly from `StateManager` must make use of its state management mechanisms explicitly (these interactions are normally undertaken by `LockManager`). From a programmer's point of view this amounts to making appropriate use of the operations `activate`, `deactivate` and `modified`, since `StateManager's` constructors are effectively identical to those of `LockManager` shown in Chapter 3.

```
boolean activate ()
boolean activate (String storeRoot)
```

`Activate` loads an object from the object store. The object's UID must already have been set via the constructor and the object must exist in the store. If the object is successfully read then `restore_state` is called to build the object in memory. `Activate` is idempotent so that once an object has been activated further calls are ignored. The parameter represents the root name of the object store to search for the object. A value of null means use the default store.

```
boolean deactivate ()
boolean deactivate (String storeRoot)
```

The inverse of `activate`. First calls `save_state` to build the compacted image of the object which is then saved in the object store. Objects are only saved if they have been modified since they were activated. The parameter represents the root name of the object store into which the object should be saved. A value of null means use the default store.

```
void modified ()
```

Must be called prior to modifying the object in memory. If it is not called the object *will not* be saved in the object store by `deactivate`.

5. Appendix A: Class library

This chapter contains an overview of those classes that the application programmer will typically use. The aim of this chapter is to provide a quick reference guide to these classes for use when writing applications in *JavaArjuna*. For clarity only the public and protected interfaces of the classes will be given.

5.1 LockManager

```
public class LockResult
{
public static final int GRANTED;
public static final int REFUSED;
public static final int RELEASED;
};

public class ConflictType
{
public static final int CONFLICT;
public static final int COMPATIBLE;
public static final int PRESENT;
};

public abstract class LockManager extends StateManager
{
public static final int defaultRetry;
public static final int defaultTimeout;
public static final int waitTotalTimeout;

public final synchronized boolean releaselock (Uid lockUid);
public final synchronized int setlock (Lock toSet);
public final synchronized int setlock (Lock toSet, int retry);
public final synchronized int setlock (Lock toSet, int retry, int sleepTime);

public void print (PrintStream strm);

public String type ();
public boolean save_state (OutputObjectState os, int ObjectType);
public boolean restore_state (InputObjectState os, int ObjectType);

protected LockManager ();
protected LockManager (int ot);
protected LockManager (int ot, ObjectName attr);
protected LockManager (Uid storeUid);
protected LockManager (Uid storeUid, int ot);
protected LockManager (Uid storeUid, int ot, ObjectName attr);

protected void terminate ();
};
```

5.2 StateManager

```
public class ObjectStatus
{
public static final int PASSIVE;
public static final int PASSIVE_NEW;
public static final int ACTIVE;
public static final int ACTIVE_NEW;
};

public class ObjectType
{
public static final int RECOVERABLE;
public static final int ANDPERSISTENT;
public static final int NEITHER;
};

public abstract class StateManager
{
public boolean restore_state (InputObjectState os, int ot);
public boolean save_state (OutputObjectState os, int ot);
public String type ();

public synchronized boolean activate ();
public synchronized boolean activate (String rootName);
public synchronized boolean deactivate ();
public synchronized boolean deactivate (String rootName);
public synchronized boolean deactivate (String rootName, boolean commit);

public synchronized int status ();
public final Uid get_uid ();
public void destroy ();
public void print (PrintStream strm);

protected void terminate ();

protected StateManager ();
protected StateManager (int ot);
protected StateManager (int ot, ObjectName objName);
protected StateManager (Uid objUid);
protected StateManager (Uid objUid, int ot);
protected StateManager (Uid objUid, int ot, ObjectName objName);

protected synchronized final void modified ();
};
```

5.3 Input/OutputObjectState

```
class OutputObjectState extends OutputBuffer
{
public OutputObjectState (Uid newUid, String typeName);

public boolean notempty ();
public int size ();
public Uid stateUid ();
public String type ();
};

class InputObjectState extends ObjectState
{
public OutputObjectState (Uid newUid, String typeName, byte[] b);

public boolean notempty ();
public int size ();
public Uid stateUid ();
public String type ();
};
```

5.4 Input/OutputBuffer

```
public class OutputBuffer
{
public OutputBuffer ();

public final synchronized boolean valid ();
public synchronized byte[] buffer();
public synchronized int length ();

       /* pack operations for standard Java types */

public synchronized void packByte (byte b) throws IOException;
public synchronized void packBytes (byte[] b) throws IOException;
public synchronized void packBoolean (boolean b) throws IOException;
public synchronized void packChar (char c) throws IOException;
public synchronized void packShort (short s) throws IOException;
public synchronized void packInt (int i) throws IOException;
public synchronized void packLong (long l) throws IOException;
public synchronized void packFloat (float f) throws IOException;
public synchronized void packDouble (double d) throws IOException;
public synchronized void packString (String s) throws IOException;
};

public class InputBuffer
{
public InputBuffer ();

public final synchronized boolean valid ();
public synchronized byte[] buffer();
```

```
public synchronized int length ();

       /* unpack operations for standard Java types */

public synchronized byte unpackByte () throws IOException;
public synchronized byte[] unpackBytes () throws IOException;
public synchronized boolean unpackBoolean () throws IOException;
public synchronized char unpackChar () throws IOException;
public synchronized short unpackShort () throws IOException;
public synchronized int unpackInt () throws IOException;
public synchronized long unpackLong () throws IOException;
public synchronized float unpackFloat () throws IOException;
public synchronized double unpackDouble () throws IOException;
public synchronized String unpackString () throws IOException;
};
```

5.5 Uid

```
public class Uid implements Cloneable
{
public Uid ();
public Uid (Uid copyFrom);
public Uid (String uidString);
public Uid (String uidString, boolean errorsOk);

public synchronized void pack (OutputBuffer packInto) throws IOException;
public synchronized void unpack (InputBuffer unpackFrom) throws IOException;

public void print (PrintStream strm);

public String toString ();

public Object clone () throws CloneNotSupportedException;
public synchronized void copy (Uid toCopy) throws UidException;

public boolean equals (Uid u);
public boolean notEquals (Uid u);
public boolean lessThan (Uid u);
public boolean greaterThan (Uid u);

public synchronized final boolean valid ();

public static synchronized Uid nullUid ();
};
```

5.6

Index