TIBCO ActiveSpaces® Transactions

Java Developer's Guide

Software Release 2.5.8
Published November 10, 2017

Two-Second Advantage®

Important Information

Contents

List of Figures

List of Tables

List of Examples

About this book

This guide describes how to program TIBCO ActiveSpaces® Transactions . It includes working code snippets for all of the TIBCO ActiveSpaces® Transactions features.

It is intended for the following types of readers:

- Java developers who want to get started developing Java applications using TIBCO ActiveSpaces® Transactions .

Related documentation

This book is part of a set of TIBCO ActiveSpaces® Transactions documentation, which also includes:

TIBCO ActiveSpaces® Transactions Installation — This guide describes how to install the TIBCO ActiveSpaces® Transactions software.

TIBCO ActiveSpaces® Transactions Quick Start — This guide describes how to quickly get started using Java IDEs to develop TIBCO ActiveSpaces® Transactions applications.

TIBCO ActiveSpaces® Transactions Architect's Guide — This guide provides a technical overview of TIBCO ActiveSpaces® Transactions .

TIBCO ActiveSpaces® Transactions Administration — This guide describes how to install, configure, and monitor an TIBCO ActiveSpaces® Transactions deployment.

TIBCO ActiveSpaces® Transactions Performance Tuning Guide — This guide describes the tools and techniques to tune TIBCO ActiveSpaces® Transactions applications.

TIBCO ActiveSpaces® Transactions System Sizing Guide — This guide describes how to size system resources for TIBCO ActiveSpaces® Transactions applications.

TIBCO ActiveSpaces® Transactions Javadoc — The reference documentation for all TIBCO ActiveSpaces® Transactions APIs.

Conventions

The following conventions are used in this book:

Bold — Used to refer to particular items on a user interface such as the **Event Monitor** button.

`Constant Width` **—** Used for anything that you would type literally such as keywords, data types, parameter names, etc.

`Constant Width Italic` **—** Used as a place holder for values that you should replace with an actual value.

Reference cluster

This document uses a reference cluster with these node names in the text and in the snippets.

- A - application node

- `B` - application node
- `C` - application node
- `domainmanager` - Distributed domain management

The management domain name is `Development`.

Code snippets

All of the code snippets in this book are self contained. They use the reference cluster defined in the section called "Reference cluster" on page xiii. All of the snippets should be run through the `domainmanager` node. Each snippet indicates which application nodes on which it should be executed.

The conventions used for the snippets are:

- All snippet package names start with `com.kabira.snippets`.
- The package name is the chapter name. For example the `com.kabira.snippets.distributedcomputing` package contains all snippets for the Distributed Computing chapter.
- Each snippet has a single public class and zero or more nested classes to support the snippet.
- The public class name is the feature or function that is being demonstrated by the snippet. For example, the `ObjectFlushing` class name is demonstrating flushing of Managed Objects.
- Each public class has a main that runs the snippet.
- No arguments are required to run the snippet.
- The node(s) on which the snippet must be executed are indicated in the comments for the snippet.
  - `domainname = Development` - run snippet on all nodes.
  - `domainnode = <node name>` - run the snippet against the specified node.

TIBCO ActiveSpaces® Transactions community

The TIBCO ActiveSpaces® Transactions online community is located at `https://devzone.tibco.com`. The online community provides direct access to other TIBCO ActiveSpaces® Transactions users and the TIBCO ActiveSpaces® Transactions development team. Please join us online for current discussions on TIBCO ActiveSpaces® Transactions development and the latest information on bug fixes and new releases.

1

Introduction

What is TIBCO ActiveSpaces® Transactions ?

TIBCO ActiveSpaces® Transactions is an in-memory transactional application platform that provides scalable high-performance transaction processing with durable object management and replication. TIBCO ActiveSpaces® Transactions allows organizations to develop highly available, distributed, transactional applications using the standard Java POJO programming model.

TIBCO ActiveSpaces® Transactions provides these capabilities:

- Transactions - high performance, distributed "All-or-None" ACID work.
- In-Memory Durable Object Store - ultra low-latency transactional persistence.
- Transactional High Availability - transparent memory-to-memory replication with instant fail-over and fail-back.
- Distributed Computing - location transparent objects and method invocation allowing transparent horizontal scaling.
- Integrated Hotspot JVM - tightly integrated Java execution environment allowing transparent low latency feature execution.

Managed objects

TIBCO ActiveSpaces® Transactions features are available using Managed Objects which provide:

- Transactions
- Distribution

- Durable Object Store
- Keys and Queries
- Asynchronous methods
- High Availability

Transactions

All TIBCO ActiveSpaces® Transactions Managed Objects are transactional. TIBCO ActiveSpaces® Transactions transactions support transactional locking, deadlock detection, and isolation. TIBCO ActiveSpaces® Transactions supports single writer, multi-reader locking, with transparent lock promotion. Deadlock detection and retry is transparently handled by the TIBCO ActiveSpaces® Transactions JVM. Transactional isolation ensures that object state modifications are not visible outside of a transaction until the transaction commits.

TIBCO ActiveSpaces® Transactions transactions can optionally span multiple JVMs on the same or different machines. Distributed locking and deadlock detection is provided.

All transactional features are native in the TIBCO ActiveSpaces® Transactions JVM and do not require any external transaction manager or database.

Durable object store

Managed Objects are always persistent in shared memory. This allows the object to live beyond the lifetime of the JVM. Shared memory Managed Objects also support extents and triggers. There is optional support for transparently integrating managed objects to a secondary store, such as an RBDMS, data grid, archival store, etc.

Keys and queries

Managed Objects can optionally have one or more keys defined. An index is maintained in shared memory for each key defined on a Managed Object. This allows high-performance queries to be performed against Managed Objects using a shared memory index.

Asynchronous methods

Asynchronous methods allow applications to queue a method for execution in a separate transaction. Transactional guarantees ensure that the method is executed once and only once in a separate transaction.

High availability

TIBCO ActiveSpaces® Transactions provides these high availability services:

- Transactional replication across one or more nodes
- Complete application transparency
- Dynamic partition definition
- Dynamic cluster membership

- Dynamic object to partition mapping
- Geographic redundancy
- Multi-master detection with avoidance and reconciliation

A partitioned Managed Object has a single active node and zero or more replica nodes. All object state modifications are transactionally completed on the current active node and all replica nodes. Replica nodes take over processing for an object in priority order when the currently active node becomes unavailable. Support is provided for restoring an object's state from a replica node during application execution without any service interruption.

Applications can read and modify a partitioned object on any node. TIBCO ActiveSpaces® Transactions transparently ensures that the updates occur on the current active node for the object. This is transparent to the application.

Partitioned Managed Objects are contained in a Partition. Multiple Partitions can exist on a single node. Partitions are associated with a priority list of nodes - the highest priority available node is the current active node for a partition. Partitions can be migrated to different nodes during application execution without any service interruption. Partitions can be dynamically created by applications or the operator.

Nodes can dynamically join and leave clusters. Active nodes, partition states, and object data is updated as required to reflect the current nodes in the cluster.

A Managed Object is partitioned by associating the object type with a Partition Mapper. The Partition Mapper dynamically assigns Managed Objects to a Partition at runtime. The Managed Object to Partition mapping can be dynamically changed to re-distribute application load across different nodes without any service interruption.

Nodes associated with a Partition can span geographies, providing support for transactionally consistent geographic redundancy across data centers. Transactional integrity is maintained across the geographies and failover and restore can occur across data centers.

Configurable multi-master, aka *split-brain*, detection is supported which allows a cluster to be either taken offline when a required node quorum is not available, or to continue processing in a non-quorum condition. Operator control is provided to merge object data on nodes that were running in a multi-master condition. Conflicts detected during the merge are reported to the application for conflict resolution.

A highly available timer service is provided to support transparent application timer notifications across failover and restore.

All high availability services are available without any external software or hardware.

Distribution

A Managed Object can be distributed. A distributed Managed Object supports transparent remote method invocation and field access. A distributed Managed Object has a single master node on which all behavior is executed at any given time. A highly available Managed Object's master node is the current active node for the partition in which it is contained. Distribution is transparent to applications.

Online cluster upgrades

Class definitions can be changed on individual nodes without requiring a cluster service outage. These class changes can include both behavior changes and object shape changes (adding, removing, changing fields). Existing objects are dynamically upgraded as nodes communicate to other nodes in the cluster. There is no impact on nodes that are running the previous version of the classes. Class changes can also be backed out without requiring a cluster service outage.

2

Developing distributed applications

This chapter describes how to develop distributed applications with TIBCO ActiveSpaces® Transactions . TIBCO ActiveSpaces® Transactions makes it easy to develop distributed applications using standard Java development tools, by transparently managing the deployment and execution of applications on multiple nodes.

The TIBCO ActiveSpaces® Transactions features provided to support distributed application development are:

- deploying applications to one or more nodes in an application domain.
- partitioning applications using domain groups within an application domain.
- dynamically adding a node to an application domain.
- automatically restoring an application to a node that is restarted in an application domain.
- application output available in the development tool for all application nodes.

Architecture

Distributed development of TIBCO ActiveSpaces® Transactions applications relies on a Domain Manager node to coordinate the deployment and execution of applications to multiple nodes.

To support distributed development the TIBCO ActiveSpaces® Transactions Deployment Tool (See the section called “Deployment tool” on page 248) is configured to connect to a Domain Manager node. The Domain Manager node coordinates all communication to the application nodes.

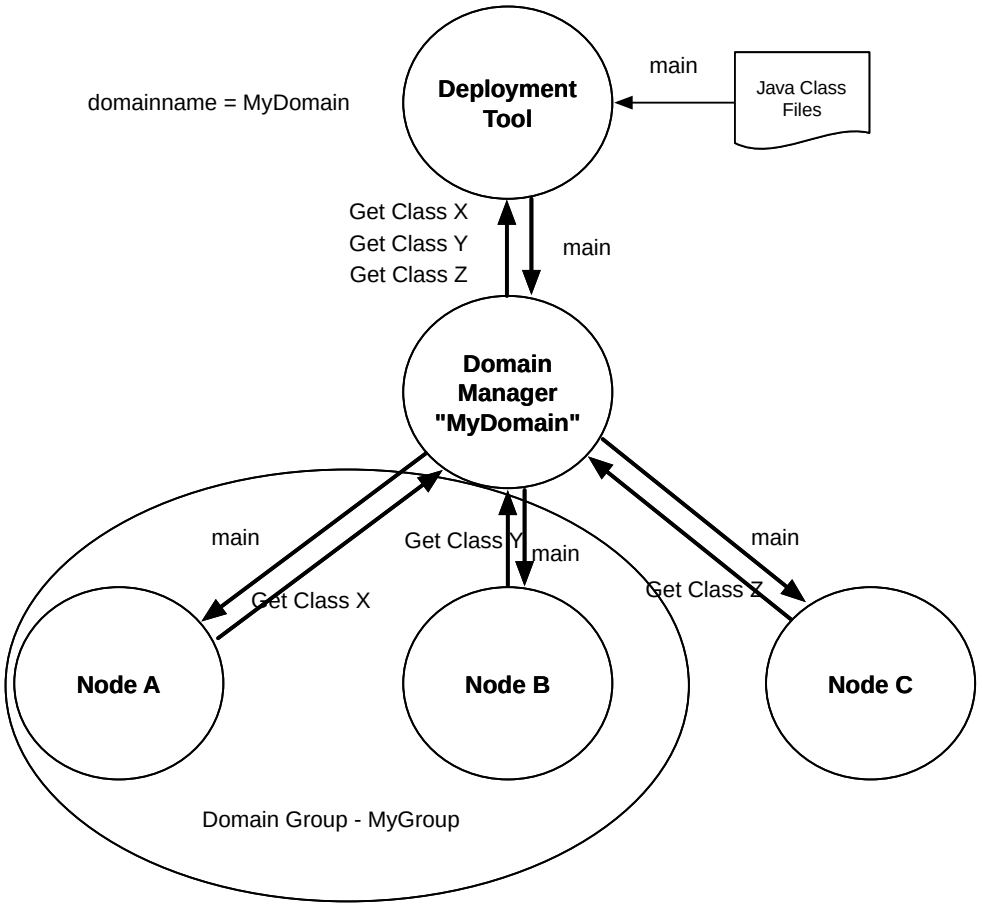

Figure 2.1. Distributed Development Architecture

When an application is executed the main entry point for the application is loaded and executed on all target nodes for the application. The same application is loaded on all application nodes. If the application requires different behavior on different nodes, application logic must provide this alternative behavior.

Once main is started on each application node, each node requests class files as needed based on application execution. This implies that different class files are executed on each node. The standard TIBCO ActiveSpaces® Transactions class resolution rules are used to locate class files. See the section called “Class resolution” on page 247 for details.

For example in Figure 2.1, Node A requests class X from the client, node B requests class Y, and node C requests class Z.

The Domain Manager monitors the execution of the application on all nodes. The deployment tool runs until all application nodes exit. Individual nodes can exit, and new ones can join the distributed application while the program is being executed.

Application scope

The application execution scope is controlled using these Deployment Tool parameters:

- `domainname` - execute the application main on all nodes in the domain.
- `domaingroup` - execute the application main on all nodes in a domain group.
- `domainnode` - execute the application main on a single node.

For example using Figure 2.1:

- `domainname = MyDomain` - executes main on Node A, Node B, and Node C.
- `domaingroup = MyGroup` - executes main on Node A and Node B.
- `domainnode = Node C` - execute mains on node C only.

The example below is run twice - once with `domainname = Development` and once with `domainnode = A`. The results are shown.

Example 2.1. Distributed Development

```
//      $Revision: 1.1.2.2 $

package com.kabira.snippets.development;

/**
 * Snippet to show program execution on multiple ActiveSpaces(R) Transactions
 * nodes
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainname</b> = Development
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class DistributedDevelopment
{
    /**
     * Main entry point
     *
     * @param args  Not used
     */
    public static void main(String [] args)
    {
                System.out.println("Welcome to ActiveSpaces Transactions");
    }
}
```

Here is the output using `domainname = Development`.

```
[A] Welcome to TIBCO ActiveSpaces® Transactions

[C] Welcome to TIBCO ActiveSpaces® Transactions

[B] Welcome to TIBCO ActiveSpaces® Transactions

INFO: Application [com.kabira.snippets.development.DistributedDevelopment] running on
node [A] exited with status [0]
INFO: Application [com.kabira.snippets.development.DistributedDevelopment] running on
node [C] exited with status [0]
INFO: Application [com.kabira.snippets.development.DistributedDevelopment] running on
node [B] exited with status [0]
INFO: Run of distributed application
[com.kabira.snippets.development.DistributedDevelopment] complete.
```

```
INFO: Application [com.kabira.snippets.development.DistributedDevelopment] exited with
status [0].
```

Here is the output using `domainnode = A`.

```
[A] Welcome to TIBCO ActiveSpaces® Transactions

INFO: Application [com.kabira.snippets.development.DistributedDevelopment1] running on
node [A] exited with status [0]
INFO: Run of distributed application
[com.kabira.snippets.development.DistributedDevelopment1] complete.
```

See the section called "Deployment example" on page 255 for details on running this example from the command line.

Adding a node

When a new node is joins a domain that is currently executing an application, the application is deployed to the new node transparently. Any application data required for that node should be highly available so that the data is available on the new node.

Restoring a node

A node can remove itself from the distributed application by leaving the domain. A node can leave a domain because it is shutdown, an error condition, or it is explicitly removed from a domain. The Deployment Tool is notified that a node left the distributed application, however, execution continues.

A node that removed itself from a distributed application can rejoin the distributed application by joining the domain again. When the node is active in the domain again, it is treated the same as a new node being added to the domain as described in the section called "Adding a node" on page 8.

Debugging

A Java debugger can be remotely attached to any of the application nodes participating in a distributed application. These Deployment Tool parameters can be used to control debugging of distributed applications:

- `remotedebug` - enable remote debug port on all target application nodes.
- `suspend` - suspend all target application nodes before executing main.

When using `suspend` to control execution of main on target application nodes, the debugger must be connected to each application node to continue application execution. If the debugger is not connected to an application node, the application will never continue executing on that node.

A complete example of attaching to a TIBCO ActiveSpaces® Transactions application with a debugger from the command line is in the section called "Debugging example" on page 257.

3

Java Virtual Machine

This chapter describes details specific to the TIBCO ActiveSpaces® Transactions Java Virtual Machine (JVM).

When an TIBCO ActiveSpaces® Transactions JVM is first started it executes the application's `main` method passing in any specified application parameters.

Shutdown

A JVM shutdown sequence, shown in Figure 3.1, is triggered by:

- all non-daemon threads, including `main`, which is executed in a non-daemon thread, completing. Any executing daemon threads do not prevent the shutdown sequence from starting.
- an application calling `Runtime.getRuntime().exit()` or `System.exit()`. Calls to the `exit()` methods do not wait for non-daemon threads to complete before starting the shutdown sequence.
- an operator shutting down the JVM, or node, using the TIBCO ActiveSpaces® Transactions Administrator. An operator initiated shutdown behaves the same as a call to an `exit()` method - the shutdown sequence is started without waiting for non-daemon threads to complete.

The shutdown sequence waits a configurable amount of time for all transactional work to complete. Generally transactional work will automatically complete, either by committing or rolling back. However, long running transactions, or transactions blocked waiting for external resources, may exceed the shutdown wait value and cause the JVM to be forced down. The node must be restarted when this occurs. To ensure that a JVM exits cleanly make sure that an application's transactions complete within the shutdown timer interval specified using the `shutdowntimeoutseconds` parameter when the JVM was deployed.

These steps are taken during the JVM shutdown sequence:

1. A shutdown timer is started to wait for all transactions to complete. The duration of the shutdown timer is specified using the `shutdowntimeoutseconds` deployment tool value.
2. Execute any JVM shutdown hooks.
3. Block new transactions from starting.
4. Wait for all active transactions to complete.

These steps are shown in Figure 3.1.

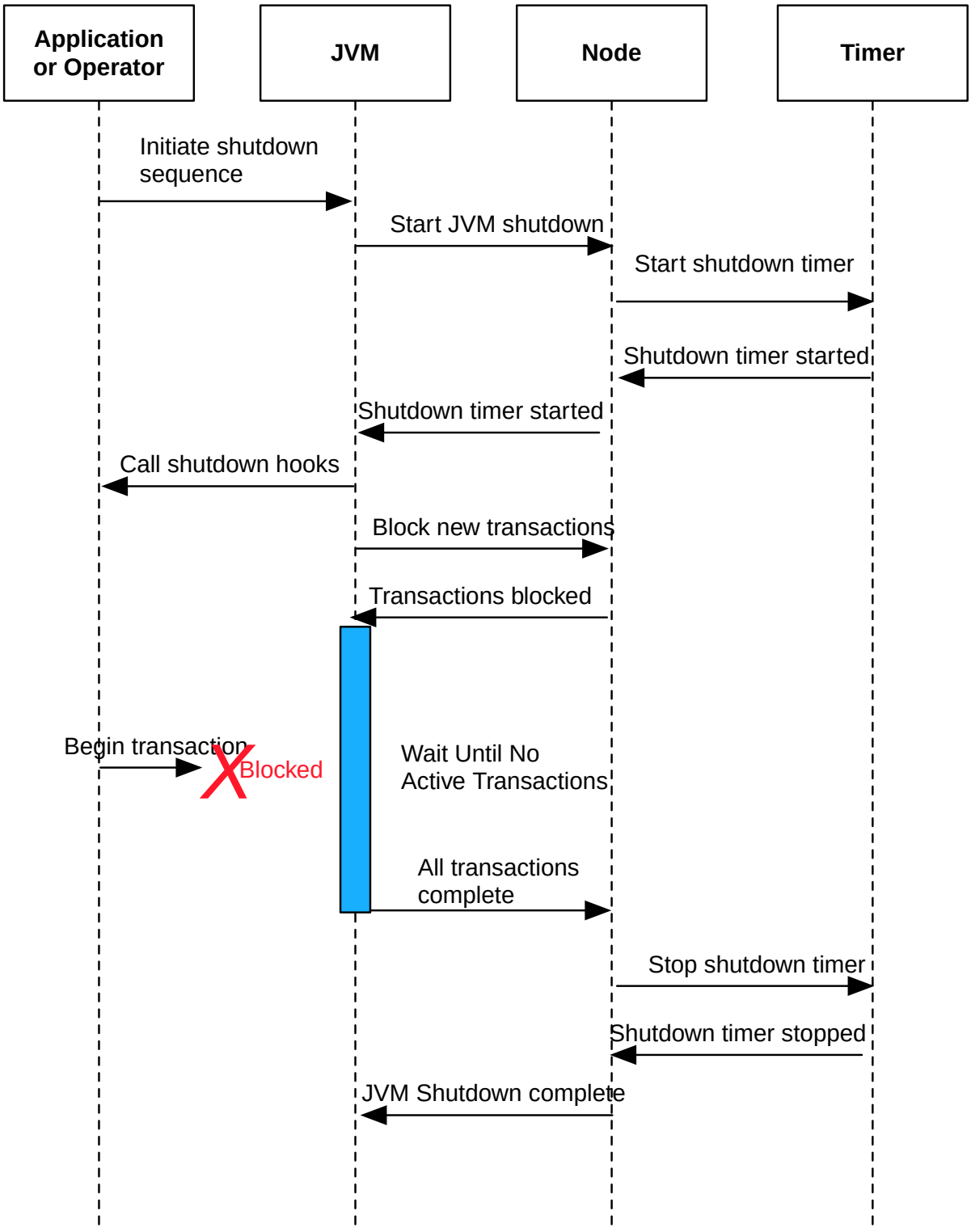

Figure 3.1. Shutdown Sequence

There is one other timer used to ensure a clean JVM termination - the target JVM resolution timer. The duration of this timer is controlled by the `noDestinationTimeoutSeconds` configuration

value. This timer controls how long a method call will block waiting for a target JVM to be available. The target JVM may be on the local or a remote node. If the method cannot be executed within the value of `noDestinationTimeoutSeconds`, the current transaction is aborted. During shutdown this transaction will not be restarted, ensuring that the JVM exits cleanly. See the **TIBCO ActiveSpaces® Transactions Administration** for details on configuring the `noDestinationTimeoutSeconds` value.

Example 3.1 on page 12 is a snippet showing the use of `exit()` to initiate a JVM shutdown sequence.

Example 3.1. Calling Exit

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.vmlifecycle;

/**
 *  Calling exit to return a non-zero return code to deployment tool
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class Exit
{
    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String args [])
    {
        //
        //    This will return a value of -1 to the deployment client
        //
        System.out.println("Calling exit with a value of -1");
        Runtime.getRuntime().exit(-1);
    }
}


#
#     Output from IDE connection if exit is called
#
INFO: Application [com.kabira.snippets.vmlifecycle.Exit] running on node [A] exited with
 status [-1]
INFO: Run of distributed application [com.kabira.snippets.vmlifecycle.Exit] complete.
FATAL: Distributed application failed on [1] nodes.
INFO: Application [com.kabira.snippets.vmlifecycle.Exit] exited with status [-1].
```

Shutdown Hooks

JVM shutdown hooks are always called during the JVM shutdown sequence. An application should use either shutdown hooks or component notifiers (see Chapter 11) to cleanly terminate an application.

It is legal to do transactional work in a shutdown hook. During JVM shutdown, a shutdown hook that is not in a transaction will not prevent the JVM from shutting down cleanly if it does not complete before other JVM shutdown tasks complete.

Managing Threads

To shut down an application, all application created non-daemon threads must exit to trigger the JVM shutdown sequence. The following approaches can be used to manage non-daemon application threads:

- Do not return from `main` until all non-daemon application threads exit.
- Signal and wait, in an application specific manner, for all non-daemon threads to exit, from a shutdown hook
- Signal and wait, in an application specific manner, for all non-daemon threads to exit, from a component notifier

Example 3.2 on page 13 shows the use of `Thread.join()` to block in `main` until all non-daemon application threads exit.

Example 3.2. Managing Threads with Join

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.vmlifecycle;

/**
 *  Using join to coordinate thread termination
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class Join
{
    /**
     * Application thread
     */
    static public class MyThread extends Thread
 {
      @Override
      public void run()
      {
           System.out.println("hello from the thread");
      }
 }
    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String[] args)
   {
          //
          //     Create and start a new thread
          //
          MyThread     t = new MyThread();

          t.run();

          //
          //     Wait for the thread to return before exiting main
          //
          try
          {
```

```
            t.join();
        }
        catch (InterruptedException ex)
        {
            // handle interrupted exception
        }

        //
        //    Returning from main - causes the JVM to exit
        //
        System.out.println("returning from main");
    }
}
```

Example 3.3 on page 14 shows how a thread can be marked as a daemon thread. Daemon threads can be used by an application if the thread termination does not need to be coordinated with the JVM being shut down.

Example 3.3. Daemon Threads

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.vmlifecycle;

/**
 *  Using daemon threads
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class Daemon
{
    /**
     * Application thread
     */
    public static class MyThread extends Thread
    {
        @Override
        public void run()
        {
            try
            {
                System.out.println("thread sleeping...");

                Thread.sleep(5000);
            }
            catch (InterruptedException ex)
            {
                //     Handle exception
            }
        }
    }

    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String[] args)
    {
        //
        //     Create a new thread
        //
        MyThread     t = new MyThread();
```

```
            //
            //     Mark the thread as a daemon thread
            //
            t.setDaemon(true);

            //
            //     Start the thread
            //
            t.run();

            //
            //     Returning from main - causes the JVM to exit
            //
            System.out.println("returning from main");
    }
}
```

System input and output

System input and output are mapped as follows in the TIBCO ActiveSpaces® Transactions JVM:

- `System.out` - `System.out` is redirected to a log file in the node directory. The log file name is the JVM name, a unique identifier, and a `.out` suffix.
- `System.in` - `System.in` is closed. Attempting to read from `System.in` will fail.

The `System.out` log file can be displayed from the TIBCO ActiveSpaces® Transactions Administrator JVM tab. It is also sent to the deployment tool client when the deployment tool is executing in attached mode. Example 3.4 on page 15 shows a simple snippet that writes and reads to system output and input.

Example 3.4. JVM input and output

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.vmlifecycle;

import java.io.IOException;

/**
 *  Standard input and output
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class InputOutput
{
    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String args []) throws IOException
    {
        System.out.println("Message to system out");

        byte [ ] buffer = new byte[100];

        //
        //  Attempt to read system input
        //
        Integer rc = System.in.read(buffer);
```

```
        System.out.println("Read return value is " + rc);
    }
}
```

When this snippet is run it outputs:

```
[A] Message to system out
[A] Read return value is -1
```

This information is also available in a file named `com_kabira_snippets_vmlifecycle_InputOutput2.out` in the node directory.

Unhandled Exceptions

Unhandled exceptions cause the current thread to exit. If the current thread is the thread in which `main` was executed, the shutdown sequence will be initiated without waiting for non-daemon threads to exit. The JVM will exit with a non-zero exit code. See the section called "Exception handling" on page 38 for unhandled exception handling when a transaction is active.

Example 3.5 on page 16 below shows an unhandled exception in the `main` thread.

Example 3.5. Unhandled Exception

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.vmlifecycle;

/**
 *  An unhandled exception in main
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class UnhandledException
{

    /**
     * Unhandled exception
     */
    public static class MyException extends java.lang.Error
    {
        /**
         * Serialization UID
         */
        public final static long serialVersionUID = 1L;
    }

    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String[] args)
    {
        //
        //    Throw an unhandled exception - non-zero exit code returned
        //    from main
        //
        throw new MyException();
    }
}
```

When Example 3.5 on page 16 is run the following output is generated:

Example 3.6. Unhandled Exception Output

```
[A] Java main class com.kabira.snippets.vmlifecycle.UnhandledException.main exited with
 an exception.
[A] Java exception occurred: com.kabira.snippets.vmlifecycle.UnhandledException$MyException
[A]         at
com.kabira.snippets.vmlifecycle.UnhandledException.main(UnhandledException.java:41)
[A]         at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
[A]         at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
[A]         at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
[A]         at java.lang.reflect.Method.invoke(Method.java:597)
[A]         at com.kabira.platform.MainWrapper.invokeMain(MainWrapper.java:47)
INFO: Application [com.kabira.snippets.vmlifecycle.UnhandledException2] running on node
 [A] exited with status [-1]
INFO: Run of distributed application [com.kabira.snippets.vmlifecycle.UnhandledException2]
 complete.
```

4

Transactions

TIBCO ActiveSpaces® Transactions provides native JVM transactions without any external databases or transaction monitors. TIBCO ActiveSpaces® Transactions transactions can be local to a JVM or span multiple JVMs on one or more TIBCO ActiveSpaces® Transactions nodes on the same or different machines.

The snippets in this section use Managed Objects which are described in Chapter 5.

Transaction boundaries

Transaction boundaries are defined using the `com.kabira.platform.Transaction` class.

An application implements the abstract `run` method to execute application code in a transaction. A transaction is implicitly started when the `execute` method is called. The `execute` method calls the application provided `run` method and executes the application code in a transaction. A transaction is terminated in the following ways:

- application code returns from the `run` method
- application throws a `Transaction.Rollback` exception from the `run` method
- a deadlock is detected (the transaction is transparently replayed)
- an unhandled exception

An application can explicitly control the outcome of a transaction by throwing the `Transaction.Rollback` exception in the `run` method. The `Transaction.Rollback` exception causes the current transaction to rollback. Returning normally from the `run` method causes the transaction to commit.

The `Transaction.Rollback` exception has two variants - with and without a `Throwable` cause.

A `Transaction.Rollback` exception that does not contain a `Throwable` cause exception is transparently caught by the `Transaction` class, the transaction is rolled back, and `Result.ROLLBACK` is returned from the `Transaction.execute()` method.

A `Transaction.Rollback` exception that contains a `Throwable` cause exception is caught by the `Transaction` class, the transaction is rolled back, and the exception is rethrown by the Transaction class as an `InvocationRunException`. The `cause` of the `InvocationRunException` is set to the `Throwable` exception that was set in the `Transaction.Rollback` exception. This provides a mechanism for application code to communicate the cause of a rollback to the caller of the `Transaction.execute()` method.

Unhandled exceptions in a `run` method cause the transaction to be rolled back. They are then rethrown unmodified.

Example 4.1 on page 20 shows a simple counting program that demonstrates a field value being rolled back.

Example 4.1. Counting example

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.transactions;

import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;

/**
 * A simple counting program showing transactional consistency
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
@Managed
class Counter
{
    int m_count = 0;
}

public class Consistency
{
    private static boolean m_commit;
    private static Counter m_counter;

    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String[] args)
    {
        //
        //  Create a counter object
        //
        new Transaction("Create Counter")
        {
            @Override
            protected void run() throws Rollback
            {
                m_counter = new Counter();
            }
        }.execute();
```

```
        //
        //  Increment the counter and commit the transaction
        //
        m_commit = true;
        incrementCounter();
        printCounter();

        incrementCounter();
        printCounter();

        //
        //  Increment the counter and rolback the transaction
        //
        m_commit = false;
        incrementCounter();
        printCounter();
    }

    private static void incrementCounter()
    {
        new Transaction("Increment Counter")
        {
            @Override
            protected void run() throws Rollback
            {
                m_counter.m_count += 1;

                if (m_commit == true)
                {
                    return;
                }
                throw new Transaction.Rollback();
            }
        }.execute();
    }

    private static void printCounter()
    {
        new Transaction("Print Counter")
        {
            @Override
            protected void run() throws Rollback
            {
                System.out.println(m_counter.m_count);
            }
        }.execute();
    }
}
```

When run, this simple program outputs (annotation added):

Example 4.2. Counting example output

```
#
#     Initial call to execute that commits
#
1


#
#     Second call to execute that commits
#
2


#
#     Third call to execute that rolls back - field restored to value before call
```

```
#
2
```

Example 4.3 on page 22 shows the use of a `Throwable` cause to communicate rollback information to the caller of the `Transaction.execute()`.

Example 4.3. Throwable cause example

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.transactions;

import com.kabira.platform.Transaction;

class Cause extends Throwable
{
    final static long serialVersionUID = 1L;

    Cause(String message)
    {
        super (message);
    }
}

/**
 * Rollback a transaction with a Throwable exception
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class RollbackWithCause
{
    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String [] args)
    {
        try
        {
            new Transaction("Rollback")
            {
                @Override
                protected void run() throws Rollback
                {
                    Cause   cause = new Cause("rollback because of error");
                    throw new Transaction.Rollback(cause);
                }
            }.execute();
        }
        catch (Transaction.InvocationRunException ex)
        {
            System.out.println("INFO: " + ex.getCause().getMessage());
        }
    }
}
```

When run, this simple program outputs:

Example 4.4. Throwable cause example output

```
[A] INFO: rollback because of error
```

Transaction isolation level

These transaction isolation levels are supported:

- Serializable (default)
- Read Committed - Snapshot

The transaction isolation level can only be set before a transaction is started. The isolation level is in effect until a transaction commits or rolls-back. Example 4.5 on page 23 demonstrates starting a serializable and a read committed - snapshot transaction.

Example 4.5. Transaction isolation level

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.transactions;

import com.kabira.platform.Transaction;
import com.kabira.platform.Transaction.Rollback;

/**
 * Transaction isolation level
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class IsolationLevel
{
    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String [] args)
    {
        Transaction.Properties properties =  new Transaction.Properties();

        //
        //  Run a serializable transaction
        //
        properties.setTransactionName("Serializable");
        properties.setIsolationLevel(
            Transaction.IsolationLevel.SERIALIZABLE);
        new Transaction()
        {
            @Override
            protected void run() throws Rollback
            {
                System.out.println(
                    getActiveTransactionProperties().getTransactionName()
                    + " using isolation level "
                    + getActiveTransactionProperties().getIsolationLevel());
            }
        }.execute(properties);

        //
        //  Run a read committed snapshot transaction
        //
        properties.setTransactionName("Read Committed - Snapshot");
        properties.setIsolationLevel(
            Transaction.IsolationLevel.READ_COMMITTED_SNAPSHOT);
```

```
        new Transaction()
        {
            @Override
            protected void run() throws Rollback
            {
                System.out.println(
                    getActiveTransactionProperties().getTransactionName()
                    + " using isolation level "
                    + getActiveTransactionProperties().getIsolationLevel());
            }
        }.execute(properties);
    }
}
```

When this snippet is run it outputs:

```
[A] Serializable using isolation level SERIALIZABLE
[A] Read Committed - Snapshot using isolation level READ_COMMITTED_SNAPSHOT
```

Transaction thread of control

Once a transaction is started all methods called from the `run` method are in the transaction. For example:

Example 4.6. Transaction thread of control

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.transactions;

import com.kabira.platform.Transaction;

/**
 * Thread of control snippet
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class ThreadOfControl
{
    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String [] args)
    {
        new Transaction("Thread of Control")
        {
            @Override
            protected void run() throws Rollback
            {
                methodOne();
            }
        }.execute();
    }

    private static void methodOne()
    {
        //
        //    This is executing in a transaction
        //
        methodTwo();
```

```
    }

    private static void methodTwo()
    {
        //
        //    This is also executing in a transaction
        //
        // ...
    }
}
```

The *thread of control* of a transaction can span threads, JVMs, and nodes. Executing a method on a remote object extends the transaction to the remote node transparently. Application programs cannot assume that a transaction executes in a single thread.

If a new thread is created in a transaction, the new thread is not executing in a transaction when it starts. The creation of a thread is also not transactional. Specifically if a thread is started in a transaction and the transaction rolls back the thread is still running.

Example 4.7. Thread creation

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.transactions;

import com.kabira.platform.Transaction;

/**
 * Starting threads in a transaction
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class Threads
{
    /**
     * An application thread
     */
    public static class MyThread extends Thread
    {
        @Override
        public void run()
        {
            System.out.println("new thread not in a transaction");
        }
    }

    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String [] args)
    {
        new Transaction("Threads")
        {
            @Override
            protected void run() throws Rollback
            {
                //
                //    Create a new daemon thread
                //
                MyThread    myThread = new MyThread();
                myThread.setDaemon(true);
```

```
                //
                //    The thread is started even if the transaction rolls back.
                //    The thread run method is not in a transaction
                //
                myThread.start();           }
        }.execute();
    }
}
```

Locking and deadlocks

Transaction locks are taken in the following cases on a Managed Object:

- Object creation - write lock taken.
- Object deletion - write lock taken.
- Field modified - write lock taken.
- Field read - read lock taken for serializable isolation level, snapshot (no read locks) taken for read committed - snapshot isolation level

Read locks are promoted to write locks if object fields are first read and then modified. Lock promotion can only occur if a serializable isolation level is being used in a transaction.

Transaction locks are held until the current transaction commits or aborts.

No locks are taken when a method is invoked. It is possible to execute a method on an object without taking any transaction locks if that method does not access any fields in the object.

Example 4.8. Object locking

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.transactions;

import com.kabira.platform.Transaction;
import com.kabira.platform.ManagedObject;
import com.kabira.platform.annotation.Managed;

/**
 * Snippet showing transaction locking
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class Locking extends Transaction
{
    @Managed
    public static class MyObject
    {
        private MyObject() { };

        /**
         * Create MyObject
         *
         * @param name  Initialize name field
         */
```

```
        public MyObject(String name)
        {
            this.name = name;
        }
        /**
         * Name
         */
        public String name;
        /**
         * Lock field
         */
        public boolean lock;
    }

    /**
     *    Transaction to create an instance of MyObject
     */
    public static class CreateTransaction extends Transaction
    {
        private MyObject m_a;

        /**
         * Transaction run method
         */
        @Override
        protected void run()
        {
            m_a = new MyObject("existing");
        }
    }

    /**
     *    Transaction to delete an instance of MyObject
     */
    public static class DeleteTransaction extends Transaction
    {
        private MyObject m_a;

        /**
         * Transaction run method
         */
        @Override
        protected void run()
        {
            //
            //  Cache a reference to the object before deleting it
            //
            MyObject myObject = m_a;
            String name = myObject.name;

            if (Transaction.hasWriteLock(myObject) == false)
            {
                System.out.println(name + ": does not have a write lock");
            }

            //
            //    Deleting an object takes a write lock
            //
            ManagedObject.delete(m_a);

            if (Transaction.hasWriteLock(myObject) == true)
            {
                System.out.println(name + ": now has a write lock");
            }
        }
    }
    private MyObject m_a;
```

```
    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String[] args)
    {
        Locking locking = new Locking();
        CreateTransaction createTransaction = new CreateTransaction();
        DeleteTransaction deleteTransaction = new DeleteTransaction();

        createTransaction.execute();

        locking.m_a = createTransaction.m_a;
        locking.execute();

        deleteTransaction.m_a = createTransaction.m_a;
        deleteTransaction.execute();
    }

    /**
     * Transaction run method
     */
    @Override
    protected void run()
    {
        MyObject a = new MyObject("created");

        if (Transaction.hasWriteLock(a) == true)
        {
            System.out.println(a.name + ": has a write lock");
        }

        //
        //    This object does not have a write lock because it was created
        //    outside of this transaction.  Reading the name field will
        //    take a read lock.
        //
        if (Transaction.hasWriteLock(m_a) == false)
        {
            System.out.println(m_a.name + ": does not have a write lock");
        }
        if (Transaction.hasReadLock(m_a) == true)
        {
            System.out.println(m_a.name + ": now has a read lock");
        }

        //
        //    Take a write lock by setting the lock attribute.  This
        //    promotes the read lock taken above when name was read.
        //
        m_a.lock = true;

        if (Transaction.hasWriteLock(m_a) == true)
        {
            System.out.println(m_a.name + ": now has a write lock");
        }
    }
}
```

When Example 4.8 on page 26 is run it generates the following output:

Example 4.9. Object locking output

```
[A] created: has a write lock
[A] existing: does not have a write lock
```

```
[A] existing: now has a read lock
[A] existing: now has a write lock
[A] existing: does not have a write lock
[A] existing: now has a write lock
```

Deadlocks are handled transparently and do not have to be explicitly handled by the application. When a deadlock occurs, the `Transaction` class detects the deadlock, rolls back the current transaction and restarts a new transaction by calling the `run` method again.

Explicit locking

It is possible to explicitly transaction lock objects. Explicit transaction locking is useful to avoid lock promotions. A lock promotion happens when an object has a read lock and then the object is modified. This is usually caused by first reading a field value and then modifying the object when using a serializable transaction isolation.

These mechanisms are available to explicitly lock objects:

- `Transaction.readLockObject` - using a serializable isolation level, read locks an object. Performs a snapshot when using read committed - snapshot isolation level.
- `Transaction.writeLockObject` - explicitly write lock an object
- `ManagedObject.extent(..., LockMode objectLock)` - explicitly lock objects during extent iteration
- explicitly lock an object during queries

The example below show how to avoid a lock promotion.

Example 4.10. Avoiding lock promotion

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.transactions;

import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;

/**
 * Explicit object locking.
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class LockPromotion extends Transaction
{
    @Managed
    public static class MyObject
    {
        String input;
        String output;
    }

    /**
     * Control program execution
     */
    public enum Action
    {
        /**
```

```
         * Create the object
         */
        CREATE,
        /**
         * Promote a read lock
         */
        PROMOTE,
        /**
         * Take a write lock - avoiding lock promotion
         */
        WRITELOCK
    }
    private MyObject m_b1;
    private Action m_action;

    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String[] args)
    {
        LockPromotion lockPromotion = new LockPromotion();

        lockPromotion.m_action = Action.CREATE;
        lockPromotion.execute();

        lockPromotion.m_action = Action.PROMOTE;
        lockPromotion.execute();

        lockPromotion.m_action = Action.WRITELOCK;
        lockPromotion.execute();
    }

    /**
     * Report locks held on objects
     * @param msg   message to include in lock report
     */
    public void reportLock(String msg)
    {
        System.out.println(msg + " B1: read lock = "
            + Transaction.hasReadLock(m_b1)
            + ", write lock = "
            + Transaction.hasWriteLock(m_b1));
    }

    /**
     * Transaction run method
     *
     * @throws com.kabira.platform.Transaction.Rollback
     */
    @Override
    protected void run() throws Rollback
    {
        if (m_action == Action.CREATE)
        {
            m_b1 = new MyObject();
        }
        else if (m_action == Action.PROMOTE)
        {
            reportLock("promote: enter");

            //
            //     Accessing input takes a read lock
            //
            String i = m_b1.input;
            reportLock("promote: read");
```

```
            //
            //     Read lock is promoted to write lock. Note this
            //     also happens when the following is executed:
            //
            //     m_b1.output = m_b1.input;
            //
            m_b1.output = i;
            reportLock("promote: write");
        }
        else
        {
            assert (m_action == Action.WRITELOCK);

            reportLock("writelock: enter");

            //
            //     Explicitly take write lock to avoid promotion
            //
            Transaction.writeLockObject(m_b1);

            //
            //     Accessing input will already have write lock
            //
            String i = m_b1.input;
            reportLock("writelock: read");

            //
            //     No promotion of locks happen
            //
            m_b1.output = i;
            reportLock("writelock: write");
        }
    }
}
```

When Example 4.10 on page 29 is run it outputs the following (annotation added):

Example 4.11. Avoiding lock promotion output

```
[A] promote: enter B1: read lock = false, write lock = false

#
#     Read lock is taken when field on B1 is read
#
[A] promote: read B1: read lock = true, write lock = false

#
#     Write lock is taken when field on B1 is set
#
[A] promote: write B1: read lock = true, write lock = true

[A] writelock: enter B1: read lock = false, write lock = false

#
#    Explicitly write lock B1 causes both the read and write lock
#    to be taken on B1
#
[A] writelock: read B1: read lock = true, write lock = true
[A] writelock: write B1: read lock = true, write lock = true
```

Interaction with Java monitors

Java monitors can deadlock with transaction locks. Figure 4.1 shows an undetected deadlock between a Java monitor and a transaction lock.

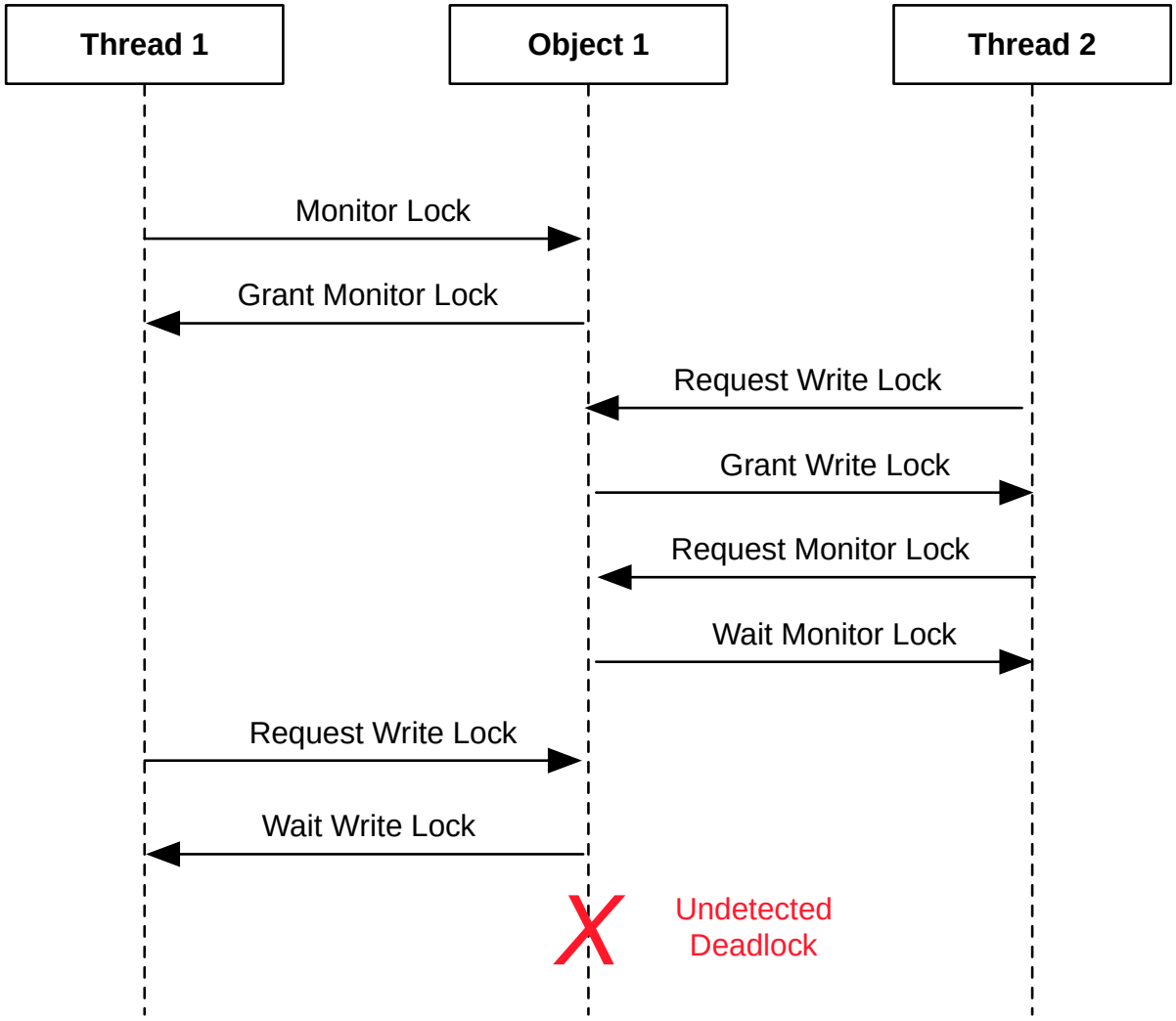

Figure 4.1. Undetected deadlock

To avoid undetected deadlocks between monitors and transaction locks, monitors should not be used with transactional resources. In general, monitors are not needed in a transactional system because transaction isolation provides the same data integrity guarantees with much better concurrency and ease of use.

Notifiers

TIBCO ActiveSpaces® Transactions supports notification of transaction resolution using the `com.kabira.platform.swbuiltin.TransactionNotifier` class. This class provides `onPrepare`, `onRollback` and `onCommit` methods that can be implemented as required to integrate with external transactional and non-transactional resources.

Multiple transaction notifiers can be created during the same transaction. The appropriate method is called for each notifier instance created as the transaction progresses. The order in which multiple notifiers is called is undefined so there should be no order assumptions in the notifiers. A notifier

that is created in a transaction can be deleted before the transaction completes. In this case, the notifier is not called when the transaction completes.

The `onPrepare` method is always called before `onCommit`. The `onPrepare` method can be used to *prepare* external XA transactional resources. `onPrepare` can report a failure by throwing an exception. If that happens, the transaction will be rolled back.

The `onPrepare`, `onCommit`, and `onRollback` methods are always invoked on the node where the transaction is executing. This is true even if the notifier is a distributed object. If other methods are called on a distributed notifier object from within the `onPrepare`, `onCommit`, or `onRollback` methods, they are dispatched to the remote node.

When a failure occurs compensation must be done to ensure that any work that was completed before the failure is restored to its initial state. TIBCO ActiveSpaces® Transactions transactional resources are automatically restored to their initial state by rolling back any changes when a transaction aborts - eliminating any need for compensation. However, non-transactional resources (e.g. a file or network connection) that are modified during a transaction, may require state changes to be explicitly restored by the application using manual compensation. The `onPrepare`, `onCommit`, and `onRollback` methods can be used by applications to implement manual compensation for non-transactional resources.

Example 4.12. Transaction notifiers

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.transactions;

import com.kabira.platform.Transaction;
import com.kabira.platform.swbuiltin.TransactionNotifier;

/**
 * Snippet showing transaction notifiers
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class Notifiers extends Transaction
{
    /**
     * Transaction notifier
     */
    static class Compensation extends TransactionNotifier
    {
        String        name;

        /*
         * Prepare notifier
         */
        @Override
        public void onPrepare()
        {
            //
            //    Perform application specific prepare processing
            //
            System.out.println(name + ": onPrepare called");
        }

        /**
         * Rollback notifier
         */
```

```
        @Override
        public void onRollback()
        {
            //
            //    Perform application specific rollback processing
            //
            System.out.println(name + ": onRollback called");

            //
            //    Do not need to call delete.  The notifier instance
            //    deletes itself.
            //
        }

        /**
         * Commit notifier
         */
        @Override
        public void onCommit()
        {
            //
            //    Perform application specific commit processing
            //
            System.out.println(name + ": onCommit called");


            //
            //    Do not need to call delete.  The notifier instance
            //    deletes itself.
            //
        }
    }
    Transaction.Result    result;

    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String [] args)
    {
        Notifiers    notifiers = new Notifiers();

        notifiers.result = Result.COMMIT;
        notifiers.execute();

        notifiers.result = Result.ROLLBACK;
        notifiers.execute();
    }

    /**
     * Transaction run method
     *
     * @throws com.kabira.platform.Transaction.Rollback
     */
    @Override
    protected void run() throws Transaction.Rollback
    {
        op1();
        op2();
        op3();

        if (result == Result.ROLLBACK)
        {
            throw new Transaction.Rollback();
        }
    }
```

```
    /**
     * Operation 1
     * <p>
     * Create a transaction notifier to compensate for this operation
     */
    public void op1()
    {
        Compensation    compensation = new Compensation();
        compensation.name = "op1";
    }

    /**
     * Operation 2
     * <p>
     * Create a transaction notifier to compensate for this operation
     */
    public void op2()
    {
        Compensation    compensation = new Compensation();
        compensation.name = "op2";
    }
    /**
     * Operation 3
     * <p>
     * Create a transaction notifier to compensate for this operation
     */
    public void op3()
    {
        Compensation    compensation = new Compensation();
        compensation.name = "op3";
    }
}
```

When Example 4.12 on page 33 is run it outputs (annotation added):

Example 4.13. Transaction notifier output

```
#
#     prepare methods call before commit
#
[A] op1: onPrepare called
[A] op2: onPrepare called
[A] op3: onPrepare called

#
#     commit compensation methods
#
[A] op1: onCommit called
[A] op2: onCommit called
[A] op3: onCommit called

#
#    no prepare methods called because transaction was explicitly rolled back in execute
#    before commit processing started
#


#
#    rollback compensation methods
#
[A] op1: onRollback called
[A] op2: onRollback called
[A] op3: onRollback called
```

Transaction notifier restrictions

The `onCommit` and `onRollback` methods in a transaction notifier have the following restrictions:

- Partitioned objects cannot be modified.
- No new transaction locks can be taken.

Attempting to modify a partitioned object in the `onCommit` or `onRollback` methods will cause a `com.kabira.platform.ResourceUnavailableException` to be thrown causing the transactional work to be discarded.

All transaction locks must be taken before the `onCommit` or `onRollback` methods are called. This restriction also precludes any Managed Objects from being created or deleted in these methods because an object create takes an implicit write lock. It is legal to take new transaction locks in the `onPrepare` method.

Incorrect usage of transaction notifiers can cause an TIBCO ActiveSpaces® Transactions node to exit with a fatal error.

Managed Objects can be deleted in the `onPrepare`, `onCommit` or `onRollback` methods if they are already write-locked.

See the section called “Explicit locking” on page 29.

The example below shows both legal and illegal object creates in the `onCommit` method.

Running Example 4.14 on page 36 with the `new Extent()` line uncommented will cause a node to exit with the following fatal exception.

```
Attempted to create a new com.kabira.snippets.transactions.Extent object from
within a commit trigger.
```

Example 4.14. Transaction notifier object locks

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.transactions;

import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.swbuiltin.TransactionNotifier;

/**
 * Snippet showing illegal transaction locking in notifiers
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class NotifierLocks
{
    /**
     * Managed objects have an extent
     */
    @Managed
    public static class Extent
    {
    };
```

```
/**
 * Non-managed objects do not have an extent
 */
public static class NoExtent
{
};

/**
 * Transaction notifier that takes an illegal lock
 */
public static class IllegalLock extends TransactionNotifier
{
    /**
     * Commit notifier
     */
    @Override
    public void onCommit()
    {
        System.out.println("IllegalLock: onCommit called");

        //
        //  Attempt to create a new extented object
        //  at commit time - this is illegal since an
        //  implicit write lock is taken on the extent.
        //
        //  Uncommenting this line will cause the Fluency node to
        //  exit with a fatal error.
        //
        // new Extent();
    }
}

/**
 * Transaction notifier that takes a legal lock
 */
public static class LegalLock extends TransactionNotifier
{
    /**
     * Commit notifier
     */
    @Override
    public void onCommit()
    {
        System.out.println("LegalLock: onCommit called");

        //
        //    Create a new object that does not have an extent.
        //    This is legal because no write locks are taken.
        //
        new NoExtent();
    }
}

/**
 * Main entry point
 * @param args  Not used
 */
public static void main(String[] args)
{
    new Transaction("Locking")
    {
        @Override
        protected void run() throws Rollback
        {
            new LegalLock();
            new IllegalLock();
```

```
            }
        }.execute();
    }
}
```

Exception handling

Unhandled exceptions in a transaction cause the current transaction to rollback and the current thread to exit. If the current thread is the thread in which main was executed, the JVM will exit. Any installed transaction notifiers are called before the thread exits (including the main thread).

Do not arbitrarily catch all exceptions in application code, only catch expected exceptions, per standard Java best practices. Unchecked exceptions are used to indicate conditions that require a transaction to be rolled back and locks released. Catching these exceptions, and not rethrowing them, will prevent transactions from being rolled back, causing unpredicatable behavior. For example, abandoned transaction locks will cause indefinite application hangs if the locked objects are attempted to be locked again.

Example 4.15 on page 38 shows an unhandled exception in the main thread.

Example 4.15. Unhandled exceptions

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.transactions;

import com.kabira.platform.Transaction;
import com.kabira.platform.swbuiltin.TransactionNotifier;

/**
 * Snippet on unhandled exceptions
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class UnhandledExceptions
{
    /**
     * An unhandled exception
     */
    public static class UnhandledException extends java.lang.Error
    {
        /**
         * Serialization UID
         */
        public final static long serialVersionUID = 1L;
    }

    /**
     * Transaction notifier
     */
    public static class Notifier extends TransactionNotifier
    {
        /**
         * Rollback notifier
         */
        @Override
        public void onRollback()
        {
            //
```

```
            //    Perform application specific rollback processing
            //
            System.out.println("onRollback called");
        }
    }
    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String [] args)
    {
        new Transaction("Unhandled Exception")
        {
            @Override
            protected void run() throws Rollback
            {
                //
                //    Create a transaction notifier
                //
                new Notifier();

                //
                //    Throw an unhandled exception - transaction rolled back
                //
                throw new UnhandledException();
            }
        }.execute();
    }
}
```

When Example 4.15 on page 38 is run it outputs (annotation added):

Example 4.16. Unhandled exception output

```
#
#     Application onRollback method called before JVM exits
#
[A] onRollback called
[A] Java main class com.kabira.snippets.transactions.UnhandledExceptions.main exited with
 an exception.
[A] com.kabira.snippets.transactions.UnhandledExceptions$UnhandledException
[A]         at
com.kabira.snippets.transactions.UnhandledExceptions.run(UnhandledExceptions.java:80)
[A]         at com.kabira.platform.Transaction.execute(Transaction.java:286)
[A]         at
com.kabira.snippets.transactions.UnhandledExceptions.main(UnhandledExceptions.java:63)
[A]         at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
[A]         at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
[A]         at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
[A]         at java.lang.reflect.Method.invoke(Method.java:597)
[A]         at com.kabira.platform.MainWrapper.invokeMain(MainWrapper.java:49)
```

Transaction required exception

Attempting to use a Managed Object object outside of a transaction will cause the following exception to be thrown:

```
java.lang.IllegalAccessError
```

Example 4.17 on page 40 will cause this exception to be thrown.

Example 4.17. Required transaction exception

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.transactions;

import com.kabira.platform.annotation.Managed;

/**
 * Snippet showing transaction required behavior
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class TransactionRequired
{
    /**
     * Transaction required class
     */
    @Managed
    public static class Required { }

    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String [] args)
    {
          //
          //     Attempting to use a transactional required class
          //     outside of a transaction
          //
          new Required();
    }
}
```

When Example 4.17 on page 40 is run it outputs (annotation added):

Example 4.18. Required Transaction Exception Output

```
#
#    java.lang.IllegalAccessError thrown because class Required requires a transaction
#
[A] Java main class com.kabira.snippets.transactions.TransactionRequired.main exited with
 an exception.
[A] Java exception occurred: java.lang.IllegalAccessError: no active transaction
[A]         at com.kabira.platform.ManagedObject._createSMObject(Native Method)
[A]         at com.kabira.platform.ManagedObject.<init>(ManagedObject.java:112)
[A]         at
com.kabira.snippets.transactions.TransactionRequired$Required.<init>(TransactionRequired.java)
[A]         at
com.kabira.snippets.transactions.TransactionRequired.main(TransactionRequired.java:45)
[A]         at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
[A]         at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
[A]         at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
[A]         at java.lang.reflect.Method.invoke(Method.java:597)
[A]         at com.kabira.platform.MainWrapper.invokeMain(MainWrapper.java:47)
```

JNI transactional programming

The Java Native Interface (JNI) is transactional if the JNI APIs are called in the context of a transaction. If there is no current transaction, no transactional behavior occurs.

This means the following when JNI is used in a transaction:

- all memory allocated, read, modified, or deleted using JNI APIs is transactional - it is logged and locked.
- the current transactional isolation is provided for field data.

All JNI code that accesses transactional resources in a transaction must check for deadlock exceptions after each call and return to the caller. This is done the same way as all other exception handling in JNI.

Example 4.19. Transactional JNI programming

```
static void JNICALL
Java_com_kabira_platform_someClass_someNative(JNIEnv *env, jclass)
{
     doSomeWork(env);

     //
     //     Check for an exception - this could be a deadlock
     //
     if (env->ExceptionCheck())
     {
           // propagate exception to caller
          return;
     }

     doMoreWork(env);

     if (env->ExceptionCheck())
     ...
}
```

Accessing an object that requires a transaction without an active transaction is not detected. This programming error will cause unpredictable behavior in the application because no transaction isolation guarantees are made for the JNI accesses.

All native resources such as file descriptors, sockets, or heap memory are always non-transactional.

Transaction notifiers can be used to support transaction safe management of non-transactional resources. The `onPrepare, onCommit,` and `onRollback` methods can be implemented as native methods to perform this management. See the section called “Notifiers” on page 32.

Transactional considerations

This section summarizes some high-level guidelines for using transactions. These are not hard and fast rules, but guidelines that should be evaluated in a specific application context.

- Avoid (or at least minimize) the use of Java monitors in transactions.

- Avoid deadlocks. When locking multiple objects always lock them in the same order. Concurrently locking objects in different orders can result in deadlocks. TIBCO ActiveSpaces® Transactions detects and handles these deadlocks transparently, but it is less expensive to avoid them.
- Avoid promotion deadlocks. When an object is going to be modified (written) within a transaction, take the write lock first, instead of the read lock. This avoids the possibility of a promotion deadlock between multiple transactions. Again TIBCO ActiveSpaces® Transactions detects and handles these deadlocks transparently, but it is less expensive to avoid them.
- Avoid resource contention. Avoid adding single points of contention to your application. If your application executes multiple threads concurrently insure that each thread uses separate resources.
- Attempt to minimize the duration of transaction locks to avoid lock contention. For example, blocking with transaction locks held waiting for data from an external source, or sleeping with transaction locks held, is generally a bad thing.
- Attempt to do all I/O outside of a transaction when the external system is non-transactional. If this cannot be avoided, use asynchronous I/O if available, to avoid blocking in a synchronous I/O operation with an active transaction.

5

Managed objects

This chapter describes how to define and use TIBCO ActiveSpaces® Transactions Managed Objects.

Defining a managed object

A Managed Object is defined using annotation. Managed Objects provide:

- shared memory persistence
- location transparent distributed access
- optionally highly available replication

Managed Objects can only be manipulated in an TIBCO ActiveSpaces® Transactions transaction. See Chapter 4 for more details on transactions.

Here is an example of a Managed Object that is persisted in shared memory.

Example 5.1. Managed object

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.managedobjects;

import com.kabira.platform.annotation.*;

/**
 *  Defining a managed object
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
@Managed
public class Manage
```

```
{
    /**
     * Name is stored in shared memory
     */
    public String    name;
}
```

No additional annotation is required for a managed object to be distributed or highly available.

@Managed annotation

The `@Managed` annotation is defined in Example 5.2 on page 44 along with its usage.

Example 5.2. @Managed annotation

```
package com.kabira.platform.annotation;

import java.lang.annotation.*;

/**
  * Marks a class as being a Managed Object - a shared memory
  * backed class. Any class that extends a Managed class is
  * also considered Managed.
  */
@Documented
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Managed
{
    /**
     * Set the allocation space reserved for this object. Default is 0,
     * which causes the system to calculate a default size.
     */
     long allocationSpaceBytes() default 0;

    /**
     * Dynamically allocate/deallocate the transaction lock memory for
     * this object, whenever it is locked in a transaction.
     * This will save approximately 112 bytes per object, when the
     * object is not locked.
     * The default is false, which causes the system to allocate the
     * lock memory once at object create time, and increases performance
     * by simplifying the work that the system needs to do whenever
     * the object is locked and unlocked.
     */
     boolean dynamicLockMemory() default false;
}

//
//  Using the @Managed annotation
//
@Managed
class Managed { ... }
```

The `@Managed` annotation is inherited by sub-types. It is legal, but not required to specify `@Managed` on sub-types.

Supported field types

The supported types for fields in Managed Objects are:

- Primitive Java types (`int`, `long`, `char`, etc.).
- Primitive wrapper classes (`Integer`, `Long`, `Character`, etc.)
- `java.lang.String`
- `java.util.Date`
- Enumerations
- Managed objects
- Non-managed objects (See the section called “Non-managed object fields” on page 45)

Fields can be arrays of these types also.

Non-managed object fields

Non-managed objects can be stored in a Managed Object field if the field is annotated with a `@ByReference` or a `@ByValue` annotation.

By-Reference Fields A managed object field annotated with the `@ByReference` annotation can store the handle to a non-managed object. A managed object field annotated with `@ByReference` is called a *by-reference field*. Example 5.3 on page 45 shows the `@ByReference` annotation definition and its use.

Example 5.3. @ByReference annotation

```
package com.kabira.platform.annotation;

/**
 * Marks a field of a non-Managed object type for inclusion in a Managed class,
 * using copy-by-reference. This allows a Managed type to safely maintain a
 * reference to process-local resources.
 */
@Documented
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface ByReference { }

//
//  Using the @ByReference annotation
//
class NotManaged { ... }

@Managed
class Managed
{
    @ByReference
    NotManaged notManaged;
}
```

By-reference fields only store local JVM handles to non-managed objects. The object state is stored in the JVM heap. Because of this, the stored objects are invalid on other JVMs. This implies that non-managed objects stored in a by-reference field are only visible in the JVM in which the field was set and they are also not replicated when they are contained by a partitioned object. By-reference fields are also not maintained across JVM shutdown.

It is legal to store different non-managed objects in the same by-reference field in different JVMs on the same, or different nodes. If a by-reference field is never set on a JVM, null is returned when the field is accessed on that JVM.

Transaction locks are taken on the managed object when a by-reference field is accessed. However, no transaction locks are taken on the non-managed object.

See the section called “Managed object life cycle” on page 51 for details on garbage collection for objects stored in by-reference fields.

By-Value Fields A managed object field annotated with the `@ByValue` annotation stores the state of a non-managed *serializable* object. A managed object field annotated with `@ByValue` is called a *by-value field*. Example 5.4 on page 46 shows the `@ByValue` annotation definition and its use. It is an error to attempt to store a non-serializable object in a by-value field.

Example 5.4. @ByValue annotation

```
package com.kabira.platform.annotation;

/**
 * Marks a Serializable field for inclusion in a Managed class,
 * using copy-by-value. This allows a Managed type to easily maintain
 * a state for a non-Managed type that can be used across VMs and nodes.
 */
@Documented
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface ByValue { }

//
//  Using the @ByValue annotation
//
class Serializable implements java.io.Serializable { ... }

@Managed
class Managed
{
    @ByValue
    Serializable serializable;
}
```

By-value fields store the serialized state of a non-managed object. Setting a by-value field performs a copy-in of the state of the non-managed object. Getting a by-value field performs a copy-out of the state of the non-managed object.

Non-managed objects stored in by-value fields are accessible in any JVM that has access to the containing managed object. The non-managed object is also replicated and is maintained across JVM shutdowns.

Transaction locks are taken on the managed object when a by-value field is accessed. However, no transaction locks are taken on the non-managed object.

Storing a non-managed object in a by-value field has no impact on when the object is garbage collected because only the object state is stored in the field, not a reference to the object itself.

Audit rules The following audit rules are enforced on the `@ByReference` and `@ByValue` annotations at class load time.

- the annotations cannot be combined on a single field.

- they cannot be specified on a static field.
- they cannot be specified on an otherwise legal managed object field type, e.g. Primitive types, `Date`, `String`, etc.

Using non-managed object fields Example 5.5 on page 47 shows the use of the `@ByReference` and `@ByValue` annotations to store non-managed objects in a managed object field.

Example 5.5. Using non-managed object fields

```
//     $Revision: 1.1.2.3 $
package com.kabira.snippets.managedobjects;

import com.kabira.platform.Transaction;
import com.kabira.platform.Transaction.Rollback;
import com.kabira.platform.annotation.ByReference;
import com.kabira.platform.annotation.ByValue;
import com.kabira.platform.annotation.Managed;

//
// A storeByValue Java object
//
class StoreByValue implements java.io.Serializable
{
    private static final long serialVersionUID = 1L;
    final String value = "serializable";
}

//
// A storeByReference heap object
//
class StoreByReference
{
    final String value = "local";
}

//
// A managed object containing a by-value and a by-reference non-managed object
//
@Managed
class Container
{
    @ByValue
    StoreByValue storeByValue;
    @ByReference
    StoreByReference storeByReference;
}

/**
 * Non-managed objects in managed object fields
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A </li>
 * </ul>
 */
public class NonManagedFields
{
    public static void main(String[] args)
    {
        new Transaction("Non-managed object fields")
        {
            @Override
            protected void run() throws Rollback
```

```
            {
                Container container = new Container();
                StoreByReference storeByReference = new StoreByReference();
                StoreByValue storeByValue = new StoreByValue();

                //
                // Store a non-managed object by-reference
                //
    container.storeByReference = storeByReference;
                StoreByReference afterByReference = container.storeByReference;

                System.out.println("INFO: by-reference before - "
                    + storeByReference.toString());
                System.out.println("INFO: by-reference after - "
                    + afterByReference.toString());

                //
                // Store a non-managed object by-value
                //
                container.storeByValue = storeByValue;
                StoreByValue afterByValue = container.storeByValue;

                System.out.println("INFO: by-value before - "
                    + storeByValue.toString());
                System.out.println("INFO: by-value after - "
                    + afterByValue.toString());
            }
        }.execute();
    }
}
```

When this snippet is run it outputs the following (annotation added):

```
//
//    Notice that the object handles are identical.  This is because
//    by-reference fields return the same object when accessed
//
[A] INFO: by-reference before -
com.kabira.snippets.managedobjects.StoreByReference@40363068
[A] INFO: by-reference after - com.kabira.snippets.managedobjects.StoreByReference@40363068

//
//    Notice that the object handle is different.  This is because
//    of the serialization copy-in/copy-out for by-value fields
//
[A] INFO: by-value before - com.kabira.snippets.managedobjects.StoreByValue@600dac21
[A] INFO: by-value after - com.kabira.snippets.managedobjects.StoreByValue@767a9224
```

Static fields

Static fields are stored locally per JVM, they are not stored in shared memory. This means that the values of static fields are different per JVM. Static field values are also not marshaled to remote nodes for distributed objects.

Example 5.6. Static fields in a managed object

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.managedobjects;

import com.kabira.platform.annotation.Managed;

/**
 *  Static fields in a managed object
 */
```

```
@Managed
class StaticFields
{
    //
    //  Store counter in non-transactional JVM local memory
    //
    private static class Data
    {
        static int counter = 0;
    }

    /**
     * Return local JVM counter
     * @return Counter value
     */
    int getCounter()
    {
        return Data.counter;
    }

    /**
     * Increment local JVM counter
     */
    void incrementCounter()
    {
        Data.counter++;
    }
}
```

Restrictions

Managed Objects have the following restrictions:

- they cannot have any `static` fields (`static final` fields are allowed).
- `finalize` cannot be implemented.
- cannot extend a non-managed class that contains non-static fields, a non-managed class with only static final fields can be extended.
- all fields must be a supported type.
- fields cannot use the `transient` modifier.
- cannot define a `serialPersistentFields` field.
- all array fields must be of a supported type.
- cannot implement the `java.io.Externalizable` interface.
- `@Managed` cannot be specified on interfaces.
- `@Managed` cannot be used on classes that extend `Enum` or `Throwable`.
- A `@Managed` class can be defined as an inner class only if it is marked `static`. The following class is legal:

```
class Outer
{
    @Managed
    public static class Inner {...}
}
```

- the size of a string field is limited by the maximum supported shared memory allocation size. This size may be found by finding the `Largest supported allocation size` in the Statistics allocator summary display.

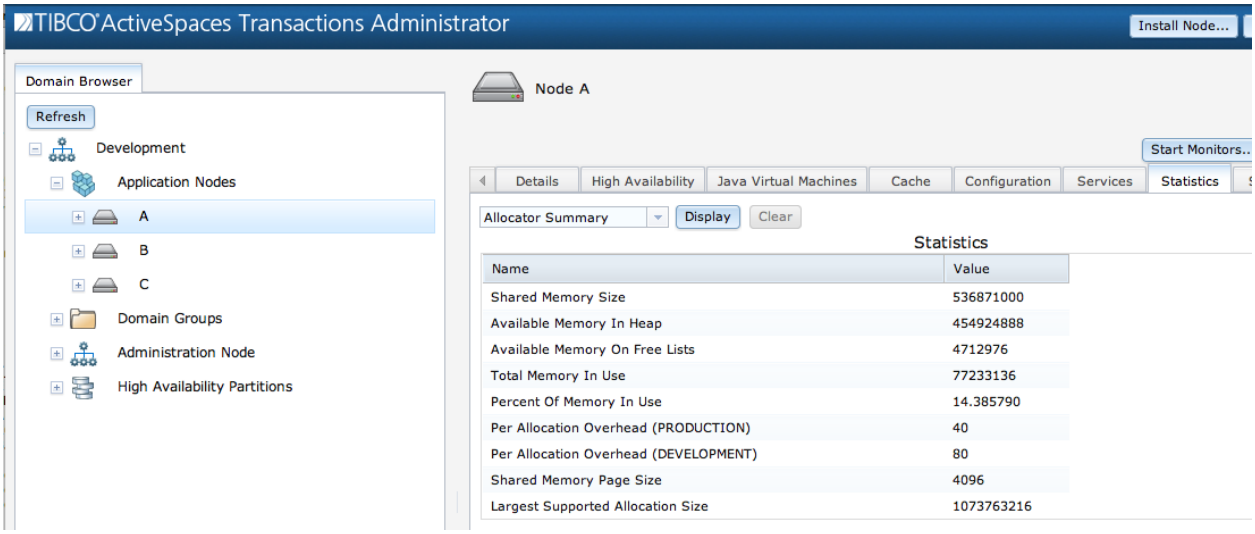

Figure 5.1. Maximum allocation size

- the number of elements in an array field of primitive types is limited by the maximum supported shared memory allocation size divided by the number of bytes in the primitive type.
- the number of elements in an array of strings field is limited by the maximum supported shared memory allocation size divided by 32.
- the number of elements in an array of arrays field is limited by the maximum supported shared memory allocation size divided by 84.
- the number of elements in an array of Managed Object fields is limited by the maximum supported shared memory allocation size divided by 24.

Managed Object restrictions are audited when the class is loaded. If any Managed Objects in a class fail audit a `java.lang.ClassNotFoundException` is thrown and the class fails to load.

Distributed method signature restrictions Managed Objects being accessed remotely (*distributed* Managed Objects) have these additional restrictions on their method signatures:

- return values must be one of the supported types (see the section called “Supported field types” on page 44), or a *serializable* object.
- method parameters must be one of the supported types (see the section called “Supported field types” on page 44), or *serializable* object.
- the elements of arrays used for method parameters must be one of the supported types (see the section called “Supported field types” on page 44), or a *serializable* object.
- the elements of arrays used for method return values must be one of the supported types (see the section called “Supported field types” on page 44), or a *serializable* object.

If a method signature does not support distribution because one of the method signature restrictions is violated, the class will load with a warning message. If the method is called on a remote object a `java.lang.UnsupportedOperationException` is thrown when the method is executed.

In addition, non-managed object method parameters cannot be null, this includes auto-box types, `String`, `Date`, and *`serializable`* objects. Passing a null non-managed object parameter causes a `java.lang.IllegalArgumentException` to be thrown when the method is called.

Managed object life cycle

All creates, reads, updates, and deletes of Managed Objects must be done in a transaction. Creating an object in a transaction that rolls back removes the object from shared memory. Deleting an object in a transaction that rolls back leaves the object in shared memory.

Managed Objects are not garbage collected by the JVM. Although the proxy Java object that references the Managed Object may be garbage collected, the shared memory state of the object remains. Managed Objects must be explicitly deleted by calling the `delete` method.

Example 5.7. Deleting managed objects

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.managedobjects;

import com.kabira.platform.Transaction;
import com.kabira.platform.ManagedObject;
import com.kabira.platform.annotation.Managed;

/**
 *  Deleting a managed object
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class Delete
{
    /**
     * A managed object
     */
    @Managed
    public static class MyObject { };

    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String [] args)
    {
        new Transaction("Delete Object")
        {
            @Override
            protected void run() throws Rollback
            {
                MyObject    e = new MyObject();

                //
                //    Delete instance in shared memory
                //
                ManagedObject.delete(e);
           }
```

```
        }.execute();
    }
}
```

After the `ManagedObject.delete` method is called on a Managed Object, using the Java reference to invoke methods or access a field will cause this exception to be thrown.

```
java.lang.NullPointerException
```

The `ManagedObject.isEmpty()` method can be used to test whether the shared memory backing a Java reference has been deleted.

If a managed object contains any by-reference fields, the non-managed object stored in the by-reference field is not eligible for garbage collection until the field is cleared, or the containing managed object is deleted.

Equals and Hashcode

If an `@Managed` class does not define an `equals()` method, one is automatically added to the class definition when the class is loaded. The generated `equals()` method will return true if two objects reference the same shared memory object.

If an `@Managed` class does not define a `hashCode()` method, one is automatically added to the class definition when the class is loaded. The generated `hashCode()` method will generate a hash value from the shared memory location of the object.

Extents

Managed Objects automatically maintain an extent. The extent makes it possible to find all instances of a Managed Object at any time. Applications should not rely on any ordering of objects in an extent.

Example 5.8. Managed object extents

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.managedobjects;

import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;

/**
 *  Using a managed object extent
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class Extent
{
    /**
     * A managed object
     */
    @Managed
    public static class MyObject
    {
        MyObject (int number)
        {
```

```
            super();
            this.number = number;
        }
        int    number;
    }

    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String [] args)
    {
        new Transaction("Extent")
        {
            @Override
            protected void run() throws Rollback
            {
                int    j;

                //
                //    Create objects in shared memory
                //
                for (j = 0; j < 10; j++)
                {
                    new MyObject(j);
                }

                //
                //    Iterate the extent deleting all of the created objects
                //
                for (MyObject myObject : ManagedObject.extent(MyObject.class))
                {
                    System.out.println(myObject.number);
                    ManagedObject.delete(myObject);
                }
           }
        }.execute();
    }
}
```

Locking and isolation

Extents support a transaction isolation of READ COMMITTED. This means that a write lock is not taken on an extent when an object is created or destroyed. This does imply that two extent iterations of the same extent in a transaction may return different results if other transactions have committed between the two iterations.

The specific extent isolation supported is:

- creates are always visible in the transaction in which they occur
- deletes are visible in the transaction in which they occur on objects that were created in the same transaction
- deletes are visible after the transaction commits on objects that were created in a separate transaction

Example 5.9 on page 54 demonstrates these rules.

Example 5.9. Extent locking

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.managedobjects;

import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
import com.kabira.platform.Transaction.Rollback;
import com.kabira.platform.annotation.Managed;

/**
 *  Extent locking
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class Locking extends Transaction
{
    /**
     * A managed object
     */
    @Managed
    public static class MyObject { }

    /**
     * Control program execution
     */
    public enum Action
    {
        /**
         * Create objects in a separate transaction
         */
        CREATE,
        /**
         * Create and delete objects in the same transaction
         */
        BOTH,
        /**
         * Delete objects in a separate transaction
         */
        DELETE
    }

    private Action    m_action;
    private String    m_message;

    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String [] args)
    {
        Locking    locking = new Locking();

        locking.m_action = Action.BOTH;
        locking.m_message = "Same Transaction";
        locking.execute();

        locking.m_action = Action.CREATE;
        locking.m_message = "Separate Transactions";
        locking.execute();

        locking.m_action = Action.DELETE;
        locking.m_message = "Separate Transactions";
```

```
        locking.execute();
    }

    /**
     * Transaction run method
     *
     * @throws com.kabira.platform.Transaction.Rollback
     */
    @Override
    protected void run() throws Rollback
    {
        int    i;

        if ((m_action == Action.BOTH) || (m_action == Action.CREATE))
        {
            for (i = 0; i < 10; i++)
            {
                new MyObject();
            }
        }

        if ((m_action == Action.BOTH) || (m_action == Action.CREATE))
        {
            System.out.println(m_message);
            System.out.println(ManagedObject.cardinality(MyObject.class)
                + " objects in extent after create");
        }

        if (m_action == Action.BOTH || m_action == Action.DELETE)
        {
            for (MyObject j1 : ManagedObject.extent(MyObject.class))
            {
                ManagedObject.delete(j1);
            }

            System.out.println(ManagedObject.cardinality(MyObject.class)
                + " objects in extent after delete");
        }
    }
}
```

When Example 5.9 on page 54 is run it outputs (annotation added):

Example 5.10. Extent locking output

```
#
#     Both creates and deletes are reflected because both are done
#     in the same transaction
#
[A] Same Transaction
[A] 10 objects in extent after create
[A] 0 objects in extent after delete

#
#     Deletes are not reflected until the tranasction commits because
#     creates occurred in a separate transction
#
[A] Separate Transactions
[A] 10 objects in extent after create
[A] 10 objects in extent after delete
```

There are two `extent` methods supported for Managed Objects - one that explicitly takes a transaction lock on returned objects and one that does not take a transaction lock on returned objects. If no transaction lock is taken on objects returned from extent iteration, a transaction lock is taken

when a field in the object is accessed or the object is explicitly locked (see the section called “Explicit locking” on page 29).

As discussed above, there is no lock taken on an extent when objects are created or deleted. The combination of no extent locking, and locks optionally not being taken on objects returned from extent iteration, may cause deleted object references being returned from an extent. This is shown in Example 5.11 on page 56.

Example 5.11. Extent object locking

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.managedobjects;

import com.kabira.platform.Transaction;
import com.kabira.platform.ManagedObject;
import com.kabira.platform.annotation.Managed;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 *  Demonstrate extent iteration with object deletion.
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class ObjectLocking
{
    /**
     * A managed object
     */
    @Managed
    public static class MyObject { };

    /**
     * Life cycle thread
     * <p>
     * This thread creates and deletes Managed Objects in shared memory.
     * Sleep to introduce some variability
     */
    public static class LifeCycleThread extends Thread
    {
        private static final int    NUMBERITERATIONS = 10;
        @Override
        public void run()
        {
            int        i;
            for (i = 0; i < NUMBERITERATIONS; i++)
            {
                try
                {
                    new LifeCycleTransaction(
      LifeCycleTransaction.Action.CREATE).execute();
                    Thread.sleep(1000);
                    new LifeCycleTransaction(
      LifeCycleTransaction.Action.DELETE).execute();
                }
                catch (InterruptedException ex)
                {
                    Logger.getLogger(
      LifeCycleThread.class.getName()).log(
      Level.SEVERE, null, ex);
                }
```

```
            }
        }
    }

    /**
     * Life cycle transaction
     * <p>
     *  Transaction to create and delete Managed Objects
     */
    public static class LifeCycleTransaction extends Transaction
    {
        private static final int    COUNT = 100;

        /**
         * Control transaction behavior
         */
        public enum Action
        {
            /**
             * Create managed objects
             */
            CREATE,
            /**
             * Delete managed objects
             */
            DELETE
        }

        LifeCycleTransaction (Action action)
        {
            m_action = action;
        }

        private Action    m_action;

        /**
         * Transaction run method
         *
         * @throws com.kabira.platform.Transaction.Rollback
         */
        @Override
        protected void run() throws Rollback
        {
            //
            //    Create managed Managed Objects
            //
            if (m_action == Action.CREATE)
            {
                int    i;

                for (i = 0; i < COUNT; i++)
                {
                    new MyObject();
                }
            }
            else
            {
                assert ( m_action == Action.DELETE );

                //
                //    Iterate extent - test for deleted objects, delete
                //    ones that are not already deleted by another thread
                //
                for (MyObject k : ManagedObject.extent(MyObject.class))
                {
                    if (ManagedObject.isEmpty(k) == false)
                    {
```

```
                        ManagedObject.delete(k);
                    }
                }
            }
        }

    }
    private static final int    NUMBERTHREADS = 15;

    /**
     * Main entry point
     * @param args  Not used
     * @throws java.lang.InterruptedException
     */
    public static void main(String [] args) throws InterruptedException
    {
        int        i;
        LifeCycleThread        threads[] = new LifeCycleThread[NUMBERTHREADS];

        for (i = 0; i < NUMBERTHREADS; i++)
        {
            threads[i] = new LifeCycleThread();
            threads[i].start();
        }

        //
        //    Wait for all of the threads to exit
        //
        for (i = 0; i < NUMBERTHREADS; i++)
        {
            threads[i].join();
        }
    }
}
```

Triggers

Triggers are defined on Managed Objects by implementing interfaces. The interfaces used are:

- `com.kabira.platform.DeleteTrigger` - add a delete trigger to the Managed Object.
- `com.kabira.platform.CompensationTrigger` - add a compensation trigger to the Managed Object.

Delete triggers are invoked when a Managed Object is deleted. The object is still valid when the delete trigger is called.

Compensation triggers are invoked on the Managed Object that caused a state conflict restoring a node following a multi-master scenario. Compensation triggers are only called on the node where the merge is occurring (see the **TIBCO ActiveSpaces® Transactions Architect's Guide**). When the compensation trigger is called, the object fields contain the data in conflict. This data will be replaced with the data from the remote node after the trigger completes execution.

Example 5.12. Triggers

```
//     $Revision: 1.1.2.3 $

package com.kabira.snippets.managedobjects;

import com.kabira.platform.DeleteTrigger;
import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
```

```
import com.kabira.platform.Transaction.Rollback;
import com.kabira.platform.annotation.Managed;

/**
 * Delete trigger
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class Triggers
{
    //
    //  Managed object that implements a delete trigger
    @Managed
    private static class MyObject implements DeleteTrigger
    {

        private MyObject() { }

        String    value;

        @Override
        public void uponDelete()
        {
            System.out.println("uponDelete: Deleting object");
        }
    }

    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String [] args)
    {
        new Transaction("Triggers")
        {
            @Override
            protected void run() throws Rollback
            {
                MyObject    myObject = new MyObject();
                ManagedObject.delete(myObject);
            }
        }.execute();
    }
}
```

When this snippet is run it outputs:

```
uponDelete: Deleting object
```

Keys and Queries

Keys are defined on Managed Objects using annotations. The annotation to define a key captures this information:

- Name of key.
- List of one or more fields making up the key.
- Property indicating whether a key is unique or non-unique.
- Property indicating whether a key is ordered or unordered.

- Property indicating whether key is mutable or not.

Here is a snippet of defining a unique unordered key on a Managed Object.

Example 5.13. Managed Object with Unique Keys

```
//     $Revision: 1.1.2.3 $
package com.kabira.snippets.managedobjects;

import com.kabira.platform.annotation.Key;
import com.kabira.platform.annotation.KeyList;
import com.kabira.platform.annotation.Managed;

/**
 * Defining a managed object with unique keys. Key fields must be declared as
 * final.
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
@Managed
@KeyList(keys =
{
    @Key(name = "ByName", fields =
    {
        "name"
    }, unique = true, ordered = false),
    @Key(name = "ByIdentifier", fields =
    {
        "identifier"
    }, unique = true, ordered = false)
})
public class Keys
{
    Keys(String name, int identifier)
    {
        this.name = name;
        this.identifier = identifier;
    }
    final String name;
    final int identifier;
}
```

Key fields are inherited from parent classes. Immutable key fields must be defined as non-static `final`. Any field type supported in Managed Objects (see the section called “Supported field types” on page 44) can be a key field. The only exceptions are:

- Array fields
- By-value field (`@ByValue` annotation)
- By-reference field (`@ByReference` annotation)

Once a key is defined on a Managed Object, an index is maintained in shared memory for that key. This index is used to perform queries against the Managed Object by specifying the key values that should be used in the query. For example queries see the section called “Queries” on page 69.

@Key and @KeyList Annotations

Two annotations are used to define keys:

- `@Key` - define a single key
- `@KeyList` - define multiple keys

A single key can be defined using the `@KeyList` annotation, but the `@Key` annotation is provided to support more concise single key definitions.

Example 5.14 on page 61 shows the definition of the `@Key` annotation and its usage.

Example 5.14. @Key Annotation

```
package com.kabira.platform.annotation;

import java.lang.annotation.*;

/** This annotation defines a single key for a Managed class.
  */
@Documented
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Key
{
    /** The name of this key on the given type. Key names must be unique
     * for a single type (including inherited keys).
     */
    String name();

    /** An ordered list of the fields that make up this key. The fields
     * must be defined in this class or in a superclass.
     */
    String[] fields();

    /** If true, the runtime will enforce that only one instance will contain
     *  the key data.
     */
    boolean unique() default true;

    /** If true, the key data will be managed as an ordered set. */
    boolean ordered() default false;

    /** If true, the key fields can be updated. */
    boolean mutable() default false;
}

//
//    Use of @Key annotation
//
@Managed
@Key
(
    name = "ByName",
    fields = { "name" },
    unique = true,
    ordered = false,
    mutable = true
)
public class Keys
{

    Keys(String name)
    {
        this.name = name;
    }
```

```
    final String name;
}
```

Example 5.15 on page 62 shows the definition of the `@KeyList` annotation and its usage.

Example 5.15. @KeyList Annotation

```
package com.kabira.platform.annotation;

import java.lang.annotation.*;

/** This annotation is used to define multiple keys on a single Managed class.
  */
@Documented
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface KeyList
{
    Key[] keys();
}

//
//    Use of @KeyList annotation
//
@Managed
@KeyList(keys=
{
    @Key(name="ByName",fields={"name"},unique=true,ordered=false),
    @Key(name="ByIdentifier",fields={"identifier"},unique=true,ordered=false)
})
public class Keys
{

    Keys(String name, int identifier)
    {
        this.name = name;
        this.identifier = identifier;
    }
    final String name;
    final int identifier;
}
```

Key Restrictions

Managed Object keys have the following restrictions:

- `@Key` and `@KeyList` cannot be used on the same class.
- `@KeyList` must be of non-zero length (at least one `@Key` must be defined).
- `@Key` and `@KeyList` can only be used on a class marked `@Managed` or a class extending a managed class.
- All fields defined in each `@Key` annotation must exist in the class, or in a superclass.
- All immutable key fields must be declared `final` and non-static.
- Mutable key fields must be declared as non-static.
- The key name must be unique within its class and among all inherited keys.

Key restrictions are audited when the class is loaded. If any key definitions in the class fail audit, a `java.lang.ClassNotFoundException` is thrown.

Inherited Keys

Key definitions are inherited by child classes. Key names must be unique in a class hierarchy. Inherited fields can be used in key definitions in child classes.

Example 5.16 on page 63 shows inherited keys being used.

Example 5.16. Inherited keys

```
//     $Revision: 1.1.4.1 $
package com.kabira.snippets.managedobjects;

import com.kabira.platform.KeyFieldValueList;
import com.kabira.platform.KeyManager;
import com.kabira.platform.KeyQuery;
import com.kabira.platform.LockMode;
import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Key;
import com.kabira.platform.annotation.Managed;

/**
 * Inherited keys
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class InheritedKeys
{
    final static String Version = "version 3.14159265359";

    @Managed
    @Key(name = "ByVersion",
    fields =
    {
        "version"
    },
    ordered = false,
    unique = false)
    private static class Base
    {
        final String version;

        Base(String version)
        {
            this.version = version;
        }

        void describe()
        {
            System.out.println(getClass().getName() + " " + this);
        }
    }

    /**
     * Extend Base - inherits the ByVersion key
     */
    private static class Extension1 extends Base
    {
        Extension1(String version)
```

```
        {
            super(version);
        }
    }

    private static class Extension2 extends Base
    {
        Extension2(String version)
        {
            super(version);
        }
    }

    public static void main(String[] args)
    {
        createObjects();
        showAllObjects();
        showExtension1Objects();
        showExtension2Objects();
    }

    static void createObjects()
    {
        new Transaction()
        {
            @Override
            public void run()
            {
                new Extension1(Version);
                new Extension1(Version);
                new Extension2(Version);
            }
        }.execute();
    }

    static void showAllObjects()
    {
        new Transaction()
        {
            @Override
            public void run()
            {
                KeyManager<Base> keyManager = new KeyManager<Base>();
                KeyQuery<Base> keyQuery = keyManager.createKeyQuery(
                Base.class, "ByVersion");

                KeyFieldValueList keyFieldValueList = new KeyFieldValueList();
                keyFieldValueList.add("version", Version);
                keyQuery.defineQuery(keyFieldValueList);

                System.out.println("Select against Base");
                for (Base obj : keyQuery.getResults(LockMode.READLOCK))
                {
                    obj.describe();
                }
            }
        }.execute();
    }

    static void showExtension1Objects()
    {
        new Transaction()
        {
            @Override
            public void run()
            {
                KeyManager<Extension1> keyManager
```

```
                = new KeyManager<Extension1>();
                KeyQuery<Extension1> keyQuery = keyManager.createKeyQuery(
                Extension1.class, "ByVersion");

                KeyFieldValueList keyFieldValueList = new KeyFieldValueList();
                keyFieldValueList.add("version", Version);
                keyQuery.defineQuery(keyFieldValueList);

                System.out.println("Select against Extension1");
                for (Extension1 obj : keyQuery.getResults(LockMode.READLOCK))
                {
                    obj.describe();
                }
            }
        }.execute();
    }

    static void showExtension2Objects()
    {
        new Transaction()
        {
            @Override
            public void run()
            {
                KeyManager<Extension2> keyManager
                = new KeyManager<Extension2>();
                KeyQuery<Extension2> keyQuery = keyManager.createKeyQuery(
                Extension2.class, "ByVersion");

                KeyFieldValueList keyFieldValueList = new KeyFieldValueList();
                keyFieldValueList.add("version", Version);
                keyQuery.defineQuery(keyFieldValueList);

                System.out.println("Select against Extension2");
                for (Extension2 obj : keyQuery.getResults(LockMode.READLOCK))
                {
                    obj.describe();
                }
            }
        }.execute();
    }
}
```

When this snippet is run it outputs (annotation added):

```
//
// Querying against base returns all instances
//
[A] Select against Base
[A] com.kabira.snippets.managedobjects.InheritedKeys$Extension2
com.kabira.snippets.managedobjects.InheritedKeys$Extension2@6733d4b8
[A] com.kabira.snippets.managedobjects.InheritedKeys$Extension1
com.kabira.snippets.managedobjects.InheritedKeys$Extension1@42bb4baf
[A] com.kabira.snippets.managedobjects.InheritedKeys$Extension1
com.kabira.snippets.managedobjects.InheritedKeys$Extension1@d3743e75

//
// Querying against Extension1 only returns Extension1 instances
//
[A] Select against Extension1
[A] com.kabira.snippets.managedobjects.InheritedKeys$Extension1
com.kabira.snippets.managedobjects.InheritedKeys$Extension1@42bb4baf
[A] com.kabira.snippets.managedobjects.InheritedKeys$Extension1
com.kabira.snippets.managedobjects.InheritedKeys$Extension1@d3743e75

//
// Querying against Extension2 only returns Extension1 instances
```

```
//
[A] Select against Extension2
[A] com.kabira.snippets.managedobjects.InheritedKeys$Extension2
com.kabira.snippets.managedobjects.InheritedKeys$Extension2@6733d4b8
```

Mutable keys

By default key values are immutable once an object has been created. However, it is possible to specify that a key value can be changed after an object has been created. This is done with the `mutable` field in the key annotation. If a key is modified after it has been created, the `KeyManager.updateIndexes(...)` method must be called after the key value is modified. If this method is not called after a key modification a runtime exception is thrown when the transaction is committed and the node is brought down.

The `KeyManager.updateIndexes(...)` method does not need to be called for key values set in a constructor.

Example 5.17 on page 66 shows how key values are modified.

Example 5.17. Modifying key values

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.managedobjects;

import com.kabira.platform.KeyFieldValueList;
import com.kabira.platform.KeyManager;
import com.kabira.platform.KeyQuery;
import com.kabira.platform.LockMode;
import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
import com.kabira.platform.Transaction.Rollback;
import com.kabira.platform.annotation.Key;
import com.kabira.platform.annotation.Managed;

/**
 * Define a managed object with a mutable key. <p> <h2> Target Nodes</h2> <ul>
 * <li> <b>domainnode</b> = A </ul>
 */
@Key(
 name = "ByName",
unique = true,
ordered = false,
mutable = true,
fields =
{
 "name"
})
@Managed
public class MutableKeys
{
 public MutableKeys(final String name)
 {
  this.name = name;
 }
 String name;

 public static void main(String[] args)
 {
  //
  // Create an instance
  //
  new Transaction("Create")
```

```
  {
   @Override
   protected void run() throws Rollback
   {
    new MutableKeys("Jim");
   }
  }.execute();

  //
  // Modify the key value
  //
  new Transaction("Modify")
  {
   @Override
   protected void run() throws Rollback
   {
    MutableKeys m = lookup("Jim");
    System.out.println("Name is " + m.name);

    //
    // Update the key value
    //
    m.name = "Bob";
    KeyManager.updateIndexes(m);
   }
  }.execute();

  //
  // Look up the object using the new key value
  //
  new Transaction("Lookup")
  {
   @Override
   protected void run() throws Rollback
   {
    MutableKeys m = lookup("Bob");
    System.out.println("Name is " + m.name);
    ManagedObject.delete(m);
   }
  }.execute();
 }

 //
 // Lookup the object
 //
 private static MutableKeys lookup(final String name)
 {
  KeyManager<MutableKeys> km = new KeyManager<MutableKeys>();
  KeyQuery<MutableKeys> kq = km.createKeyQuery(
  MutableKeys.class, "ByName");
  KeyFieldValueList fields = new KeyFieldValueList();
  fields.add("name", name);
  kq.defineQuery(fields);

  return kq.getSingleResult(LockMode.READLOCK);
 }
}
```

When this snippet is run it outputs:

```
[A] Name is Jim
[A] Name is Bob
```

Duplicate Keys

It is not possible to create multiple instances of a uniquely keyed Managed Object. Creating a second instance of a uniquely keyed Managed Object will cause this exception:

```
com.kabira.platform.ObjectNotUniqueError
```

When a duplicate key exception is thrown it indicates that the object was not created.

See the section called “Atomic Create or Select” on page 82 for the safe way to create uniquely keyed objects in the TIBCO ActiveSpaces® Transactions multi-threaded environment.

Example 5.18 on page 68 shows the handling of a duplicate key exception.

Example 5.18. Duplicate Key Exception

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.managedobjects;

import com.kabira.platform.ManagedObject;
import com.kabira.platform.ObjectNotUniqueError;
import com.kabira.platform.Transaction;
import com.kabira.platform.Transaction.Rollback;
import com.kabira.platform.annotation.Key;
import com.kabira.platform.annotation.Managed;

/**
 * Duplicate keys
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class DuplicateKey
{
    /**
     * A managed object with a unique key
     */
    @Managed
    @Key
    (
        name = "ByNumber",
        fields = { "number" },
        unique = true,
        ordered = false
    )
    public static class MyObject
    {

        /**
         * Create MyObject
         *
         * @param number    Initialize number field
         */
        public MyObject(int number)
        {
            this.number = number;
        }

        final int       number;
    }
```

```
    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String [] args)
    {
        new Transaction("Duplicate Object")
        {
            @Override
            protected void run() throws Rollback
            {
                //
                //  Create object with a key value of 1
                //
                MyObject    myObject = new MyObject(1);

                //
                //  Create another object with the same key value
                //
                try
                {
                    new MyObject(1);
                }
                catch (ObjectNotUniqueError ex)
                {
                    System.out.println(ex.getMessage());
                }

                ManagedObject.delete(myObject);
            }
        }.execute();
    }
}
```

When this snippet is run it outputs:

```
[A] Duplicate found for key
'com.kabira.snippets.managedobjects.DuplicateKey$MyObject::ByNumber'
          in com.kabira.snippets.managedobjects.DuplicateKey$MyObject, instance
1247018:5286456:2001:2.
         Duplicate is com.kabira.snippets.managedobjects.DuplicateKey$MyObject, instance
 1247018:5286456:2001:1.
          Key data:
[A] [
[A]
[A]         number = 1
[A] ]
```

Queries

The following classes provide the query interface for managed objects.

- `com.kabira.platform.KeyManager<T>` - Factory to create query objects.
- `com.kabira.platform.KeyQuery<T>` - Query object. Used to define and execute queries.
- `com.kabira.platform.QueryScope` - Support for a user-defined query scope.
- `com.kabira.platform.KeyFieldValueList` - Used to define key name/value pairs.
- `com.kabira.platform.KeyFieldValueRangeList` - Used to define key name/value pairs that have a range.

These classes are generally used in this order to define and execute a query.

1. Create a `KeyManager<T>` object.
2. Create a `KeyQuery<T>` object.
3. Create a `KeyFieldValueList` object.
4. Add key name/value pairs to the `KeyFieldValueList` object to define the query values.
5. Define the query using the `KeyQuery<T>` and the `KeyFieldValueList` objects.
6. Optionally define a user-defined query scope using a `QueryScope` object.
7. Execute the query using the `KeyQuery<T>` object.

Once a query is defined, it can be used multiple times.

Locking and Isolation Indexes defined by keys support a transaction isolation of SERIALIZABLE. This means that all updates to an index are transactionally isolated from all other transactions. Any modifications to an index caused by the creation or deletion of a keyed Managed Object is not visible outside of the current transaction until it commits.

The specific isolation rules for indexes are as follows:

- Creates are always visible in the transaction in which they occur
- Deletes are always visible in the transaction in which they occur
- Creates are visible outside of the transaction in which they were executed after the transaction commits
- Deletes are visible outside of the transaction in which they were executed after the transaction commits

The locking of objects returned from queries is explicit. When a query is executed the specific locking that should be applied to the objects returned from the query can be specified in the query API.

Non-cached remote objects returned from a query are always write locked and created in the current transaction, because they must be created on the local node before returning the result set. See Example 5.19 on page 71 for an example.

Query Scope Queries are executed on the local node, a user-defined subset of nodes in the cluster, or all active nodes in a cluster. By default queries are executed on the local node and will only return objects in the local node's index.

When a query is executed on multiple nodes, the application must be running on all nodes on which the query is executed. Remote queries executed on nodes on which the application is not running will hang until the application is started.

Query scope is supported for:

- All queries using a key, including atomic creates or select (see the section called “Atomic Create or Select” on page 82).
- Extents

- Extent cardinality

The query scope is defined using the `com.kabira.platform.QueryScope` class. As mentioned, the possible query scopes are:

- `QueryScope.QUERY_CLUSTER` - all active nodes in the cluster.
- `QueryScope.QUERY_LOCAL` - the local node only, the default query scope.
- A user-defined subset of cluster nodes.

When a user-defined query scope is defined using the `QueryScope` class, the following audits can be specified using the `QueryScope.AuditMode` enumeration:

- `AuditMode.AUDIT_NODE_LIST` - Audit that at least one node is defined in the query scope. No other audits are performed. Inactive nodes are ignored when the query is performed.
- `AuditMode.AUDIT_DISTRIBUTION` - Audit that at least one node is defined in the query scope and that distribution is active. Inactive nodes are ignored when the query is performed.
- `AuditMode.AUDIT_NODES_ACTIVE` - Audit that at least one node is defined in the query scope, distribution is active, and all nodes are active. An error is reported if there inactive nodes are included in the query scope when the query is performed.

Query scopes are defined using the following methods:

- `ManagedObject.extent(...)` for extent queries.
- `ManagedObject.cardinality(...)` for extent cardinality.
- `KeyQuery.setQueryScope(...)` for all key queries.

Example 5.19 on page 71 demonstrates a cluster wide extent query.

Example 5.19. Cluster Extent Query

```
//     $Revision: 1.1.2.4 $
package com.kabira.snippets.managedobjects;

import com.kabira.platform.LockMode;
import com.kabira.platform.ManagedObject;
import com.kabira.platform.QueryScope;
import com.kabira.platform.Transaction;
import com.kabira.platform.Transaction.Rollback;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.property.Status;

/**
 * Distributed query
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainname</b> = Development
 * </ul>
 */
public class DistributedQuery
{
    @Managed
    private static class Node
    {
```

```
        public Node(String name)
        {
            this.name = name;
        }

        final String name;
    }

    /**
     * Run the distributed query snippet
     * @param args None supported
     */
    public static void main(String [ ] args) throws InterruptedException
    {
        //
        //  Create a node object in the local node
        //
        new Transaction("Create Node Objects")
        {
            @Override
            protected void run() throws Rollback
            {
                new Node(System.getProperty(Status.NODE_NAME));
            }
        }.execute();

 //
 // Sleep some to let other nodes run
 //
 Thread.sleep(1000);

        //
        //  Find all node objects created in the cluster
        //
        //  N.B. - an indeterminate number of nodes will
        //  be returned from the extent query in this snippet
        //  because there is no synchronization of the object creation
        //  above across the nodes.
        //
        new Transaction("Query Objects")
        {
            @Override
            protected void run() throws Rollback
            {
                for (Node n : ManagedObject.extent(
                    Node.class,
                    QueryScope.QUERY_CLUSTER,
                    LockMode.READLOCK))
                {
                    System.out.println("INFO: Node " + n.name + lockType(n));
                }
            }
        }.execute();
    }

private static String lockType(final Node node)
{
 String lockType = " (";

 if (Transaction.hasReadLock(node) == true)
 {
  lockType += "READ";
 }

 if (Transaction.hasWriteLock(node) == true)
 {
  lockType += " WRITE";
```

```
  }

  if (Transaction.createdInTransaction(node) == true)
  {
   lockType += " CREATED";
  }

  lockType += ")";

  return lockType;
 }
}
```

When this snippet is run it outputs:

```
//
//   Notice that the local objects are only READ locked
//   Objects read from remote nodes are both WRITE locked and
//   CREATED in the current transaction
//
[B] INFO: Node B (READ)
[B] INFO: Node C (READ WRITE CREATED)
[B] INFO: Node A (READ WRITE CREATED)

[A] INFO: Node B (READ WRITE CREATED)
[A] INFO: Node A (READ)
[A] INFO: Node C (READ WRITE CREATED)

[C] INFO: Node B (READ WRITE CREATED)
[C] INFO: Node C (READ)
[C] INFO: Node A (READ WRITE CREATED)
```

Unique Example 5.20 on page 73 demonstrates how to define and use a single unique key to perform a query.

Example 5.20. Unique Query

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.managedobjects;

import com.kabira.platform.KeyFieldValueList;
import com.kabira.platform.KeyManager;
import com.kabira.platform.KeyQuery;
import com.kabira.platform.LockMode;
import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
import com.kabira.platform.Transaction.Rollback;
import com.kabira.platform.annotation.Key;
import com.kabira.platform.annotation.Managed;

/**
 * Unique key query
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class UniqueKey
{
    /**
     * A managed object with a unique key
     */
    @Managed
    @Key
```

```
    (
        name = "ByNumber",
        fields = { "number" },
        unique = true,
        ordered = false
    )
    private static class MyObject
    {

        /**
         * Create MyObject
         *
         * @param number        Initialize number field
         * @param description   Initialize description field
         */
        public MyObject(int number, String description)
        {
            this.number = number;
            this.description = description;
        }

        final int       number;
        final String    description;
    }

    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String [] args)
    {
        new Transaction("Unique Key")
        {
            @Override
            protected void run() throws Rollback
            {
                //
                //  Create some keyed objects
                //
                new MyObject(1, "one");
                new MyObject(2, "two");

                //
                //  Create required query objects
                //
                KeyManager<MyObject>    keyManager = new KeyManager<MyObject>();
                KeyQuery<MyObject>      keyQuery = keyManager.createKeyQuery(
                                                    MyObject.class, "ByNumber");
                KeyFieldValueList       keyFieldValueList = new KeyFieldValueList();

                //
                //  Define the query to find object 1
                //
                keyFieldValueList.add("number", 1);
                keyQuery.defineQuery(keyFieldValueList);
                System.out.println(keyQuery);

                //
                //  Perform the query - the returned object has a write lock
                //
                MyObject    myObject = keyQuery.getSingleResult(LockMode.WRITELOCK);
                System.out.println("\t" + myObject.description);
                ManagedObject.delete(myObject);

                //
                //  Redefine the query to find object 2.  Clear the previous value
                //  before reusing the keyFieldValueList
```

```
                //
                keyFieldValueList.clear();
                keyFieldValueList.add("number", 2);
                keyQuery.defineQuery(keyFieldValueList);
                System.out.println(keyQuery);

                //
                //  Perform the query - the returned object has a write lock
                //
                myObject = keyQuery.getSingleResult(LockMode.WRITELOCK);
                System.out.println("\t" + myObject.description);
                ManagedObject.delete(myObject);
            }
        }.execute();
    }
}
```

When this snippet is run it outputs:

```
[A] select obj from com.kabira.snippets.managedobjects.UniqueKey$MyObject
      using ByNumber where (number == 1); Java constructor: (none)
[A]   one
[A] select obj from com.kabira.snippets.managedobjects.UniqueKey$MyObject
      using ByNumber where (number == 2); Java constructor: (none)
[A]   two
```

Non-Unique Example 5.21 on page 75 demonstrates how to define and use a single non-unique key to perform a query. Non-unique keys must use a for loop to iterate over all returned instances.

Example 5.21. Non-Unique Query

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.managedobjects;

import com.kabira.platform.KeyFieldValueList;
import com.kabira.platform.KeyManager;
import com.kabira.platform.KeyQuery;
import com.kabira.platform.LockMode;
import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
import com.kabira.platform.Transaction.Rollback;
import com.kabira.platform.annotation.Key;
import com.kabira.platform.annotation.Managed;

/**
 * Non-unique key query
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class NonUniqueKey
{
    /**
     * A managed object with a non-unique key
     */
    @Managed
    @Key(name = "ByGroup",
    fields =
    {
        "group"
    },
    unique = false,
    ordered = false)
    private static class MyObject
```

```
    {
        /**
         * A non-unique, unordered object
         * @param group   Group key
         * @param description Description
         */
        public MyObject(int group, String description)
        {
            this.group = group;
            this.description = description;
        }
        final int group;
        final String description;
    }

    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String[] args)
    {
        new Transaction("Non-Unique Key")
        {
            @Override
            protected void run() throws Rollback
            {
                //
                //  Create some keyed objects.  Notice that a duplicate
                //  key value is specified.
                //
                new MyObject(1, "first one");
                new MyObject(1, "second one");
                new MyObject(2, "third one");

                //
                //  Create required query objects
                //
                KeyManager<MyObject> keyManager = new KeyManager<MyObject>();
                KeyQuery<MyObject> keyQuery = keyManager.createKeyQuery(
                    MyObject.class, "ByGroup");
                KeyFieldValueList keyFieldValueList = new KeyFieldValueList();

                //
                //  Define the query to find all objects with a key value of 1
                //
                keyFieldValueList.add("group", 1);
                keyQuery.defineQuery(keyFieldValueList);
                System.out.println(keyQuery);

                //
                //  Perform the query - we need to use a for loop to iterate over
                //  the objects.  We take a write lock as we execute the query.
                //
                for (MyObject myObject : keyQuery.getResults(LockMode.WRITELOCK))
                {
                    System.out.println("\t" + myObject.description);
                    ManagedObject.delete(myObject);
                }
            }
        }.execute();
    }
}
```

When this snippet is run it outputs:

```
[A] for obj in com.kabira.snippets.managedobjects.NonUniqueKey$MyObject
      using ByGroup where (group == 1) { }
```

```
[A]   second one
[A]   first one
```

Ordered

Ordered indexes allow sorting of selected objects in either ascending or descending order. They also support finding the minimum and maximum values in an index. Ordered queries can be done on unique and non-unique keys. They are also supported on multi-field keys. This allows the result set to be selectively ordered based on partial key values.

Table 5.1 on page 77 summarizes the sort order for all supported Managed Object field types.

Lexicographical sorting is only supported for single-byte character sets. Multi-byte characters are treated as single-byte characters when sorting.

Table 5.1. Sort order

Type	Sort Order
`boolean` and `Boolean`	`false` before `true`
`char` and `Character`	Lexicographical order.
`byte` and `Byte`	Numeric order.
Integer Types (`short` and `Short`, `int` and `Integer`, `long` and `Long`)	Numeric order.
Floating Point Types (`float` and `Float`, `double` and `Double`)	Numeric order.
`java.lang.String`	Lexicographical order.
`java.lang.Date`	Date order.
Enumerations	Enumeration ordinal value order.
Managed object references	Undefined.

Example 5.22 on page 77 demonstrates how to define a multi-field unique key to perform ordered queries.

Example 5.22. Ordered Query

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.managedobjects;

import com.kabira.platform.KeyFieldValueList;
import com.kabira.platform.KeyManager;
import com.kabira.platform.KeyOrderedBy;
import com.kabira.platform.KeyQuery;
import com.kabira.platform.LockMode;
import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
import com.kabira.platform.Transaction.Rollback;
import com.kabira.platform.annotation.Key;
import com.kabira.platform.annotation.Managed;

/**
 * Unique ordered key query
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
```

```
public class Ordered
{
    /**
     * A managed object with a unique multi-part key
     */
    @Managed
    @Key(name = "ByGroupDescription",
    fields =
    {
        "group", "description"
    },
    unique = true,
    ordered = true)
    private static class MyObject
    {
        /**
         * A non-unique ordered object
         * @param group   Group
         * @param description Description
         */
        public MyObject(int group, String description)
        {
            this.group = group;
            this.description = description;
        }
        final int group;
        final String description;
    }

    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String[] args)
    {
        new Transaction("Ordered Query")
        {
            @Override
            protected void run() throws Rollback
            {
                //
                //  Create some keyed objects.  These objects are unique because
                //  both the group and description field are part of the key
                //
                new MyObject(1, "a");
                new MyObject(1, "b");
                new MyObject(1, "c");
                new MyObject(2, "d");
                new MyObject(2, "e");
                new MyObject(2, "f");
                //
                //  Create required query objects
                //
                KeyManager<MyObject> keyManager = new KeyManager<MyObject>();
                KeyQuery<MyObject> keyQuery = keyManager.createKeyQuery(
                    MyObject.class, "ByGroupDescription");
                KeyFieldValueList keyFieldValueList = new KeyFieldValueList();

                //
                //  Define the query to find all objects with a key value of 1
                //  Notice that no field value is specified for the description
                //  field.  This will cause all objects with a group value of
                //  1 to be sorted on the description field.
                //
                keyFieldValueList.add("group", 1);
                keyQuery.defineQuery(keyFieldValueList);
                System.out.println(keyQuery);
```

```
//
//  Get the maximum value in group 1
//
System.out.println("Maximum Group 1 Value:");
MyObject mo = keyQuery.getMaximumResult(LockMode.READLOCK);
System.out.println("\t" + mo.description);

//
//  Get the minimum value in group 1
//
System.out.println("Minimum Group 1 Value:");
mo = keyQuery.getMinimumResult(LockMode.READLOCK);
System.out.println("\t" + mo.description);

//
//  Perform an ordered descending query on group 1 taking
//  a write lock
//
System.out.println("Descending Group 1 Query:");
for (MyObject myObject : keyQuery.getResults(
    KeyOrderedBy.DESCENDING, LockMode.WRITELOCK))
{
    System.out.println("\t" + myObject.description);
}

//
//  Perform an ordered ascending query on group 1 taking
//  a write lock
//
System.out.println("Ascending Group 1 Query:");
for (MyObject myObject : keyQuery.getResults(
    KeyOrderedBy.ASCENDING, LockMode.WRITELOCK))
{
    System.out.println("\t" + myObject.description);
}

//
//  Repeat the queries for Group 2
//
keyFieldValueList.clear();
keyFieldValueList.add("group", 2);
keyQuery.defineQuery(keyFieldValueList);
System.out.println(keyQuery);

System.out.println("Maximum Group 2 Value:");
mo = keyQuery.getMaximumResult(LockMode.READLOCK);
System.out.println("\t" + mo.description);


System.out.println("Minimum Group 2 Value:");
mo = keyQuery.getMinimumResult(LockMode.READLOCK);
System.out.println("\t" + mo.description);

System.out.println("Descending Group 2 Query:");
for (MyObject myObject : keyQuery.getResults(
    KeyOrderedBy.DESCENDING, LockMode.WRITELOCK))
{
    System.out.println("\t" + myObject.description);
}

System.out.println("Ascending Group 2 Query:");
for (MyObject myObject : keyQuery.getResults(
    KeyOrderedBy.ASCENDING, LockMode.WRITELOCK))
{
    System.out.println("\t" + myObject.description);
}
```

```
                //
                //  Remove all of the objects
                //
                for (MyObject myObject : ManagedObject.extent(MyObject.class))
                {
                    ManagedObject.delete(myObject);
                }
            }
        }.execute();
    }
}
```

When this snippet is run it outputs:

```
[A] for obj in com.kabira.snippets.managedobjects.Ordered$MyObject
    using ByGroupDescription where (group == 1) { }
[A] Maximum Group 1 Value:
[A]   c
[A] Minimum Group 1 Value:
[A]   a
[A] Descending Group 1 Query:
[A]   c
[A]   b
[A]   a
[A] Ascending Group 1 Query:
[A]   a
[A]   b
[A]   c
[A] for obj in com.kabira.snippets.managedobjects.Ordered$MyObject
    using ByGroupDescription where (group == 2) { }
[A] Maximum Group 2 Value:
[A]   f
[A] Minimum Group 2 Value:
[A]   d
[A] Descending Group 2 Query:
[A]   f
[A]   e
[A]   d
[A] Ascending Group 2 Query:
[A]   d
[A]   e
[A]   f
```

Range

The snippet shows how to perform range queries on an ordered index.

Example 5.23. Range Query

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.managedobjects;

import com.kabira.platform.KeyComparisonOperator;
import com.kabira.platform.KeyFieldValueRangeList;
import com.kabira.platform.KeyManager;
import com.kabira.platform.KeyOrderedBy;
import com.kabira.platform.KeyQuery;
import com.kabira.platform.LockMode;
import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
import com.kabira.platform.Transaction.Rollback;
import com.kabira.platform.annotation.Key;
import com.kabira.platform.annotation.Managed;
```

```
/**
 * Range query
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class Range
{
    /**
     * A managed object with a unique multi-part key
     */
    @Managed
    @Key
    (
        name = "ByNumberDescription",
        fields = { "number", "description" },
        unique = true,
        ordered = true
    )
    private static class MyObject
    {

        /**
         * Create MyObject
         *
         * @param number        Initialize number field
         * @param description   Initialize description field
         */
        public MyObject(int number, String description)
        {
            this.number = number;
            this.description = description;
        }

        final int       number;
        final String    description;
    }

    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String [] args)
    {
        new Transaction("Range Query")
        {
            @Override
            protected void run() throws Rollback
            {
                //
                //  Create some keyed objects.  These objects are unique because
                //  both the number and description field are part of the key
                //
                new MyObject(1, "a");
                new MyObject(1, "b");
                new MyObject(1, "c");
                new MyObject(2, "a");
                new MyObject(2, "b");
                new MyObject(3, "a");
                new MyObject(4, "a");
                new MyObject(5, "a");

                //
                //  Create required query objects
```

```
                //
                KeyManager<MyObject>    keyManager = new KeyManager<MyObject>();
                KeyQuery<MyObject>      keyQuery = keyManager.createKeyQuery(
                                                MyObject.class, "ByNumberDescription");
                KeyFieldValueRangeList  keyFieldValueRangeList =
                                                new KeyFieldValueRangeList();

                //
                //  Define the query to find all objects with a number value < 3
                //
                keyFieldValueRangeList.add("number", 3, KeyComparisonOperator.LT);
                keyQuery.defineQuery(keyFieldValueRangeList);

                System.out.println(keyQuery);
                for (MyObject myObject : keyQuery.getResults(LockMode.NOLOCK))
                {
                    System.out.println(
                        "\t" + myObject.number + ":" + myObject.description);
                }

                //
                //  Define the query to find all objects with a number between 1 and 5,
                //  then sort them in descending order
                //
                keyFieldValueRangeList.clear();
                keyFieldValueRangeList.add("number", 1, KeyComparisonOperator.GT);
                keyFieldValueRangeList.add("number", 5, KeyComparisonOperator.LT);
                keyQuery.defineQuery(keyFieldValueRangeList);

                System.out.println(keyQuery);
                for (MyObject myObject : keyQuery.getResults(
                    KeyOrderedBy.DESCENDING, LockMode.NOLOCK))
                {
                     System.out.println(
                         "\t" + myObject.number + ":" + myObject.description);
                }

                for (MyObject myObject : ManagedObject.extent(MyObject.class))
                {
                    ManagedObject.delete(myObject);
                }
            }
        }.execute();
    }
}
```

When this snippet is run it outputs:

```
[A] for obj in com.kabira.snippets.managedobjects.Range$MyObject
     using ByNumberDescription where (number < 3) { }
[A]  1:a
[A]  1:b
[A]  1:c
[A]  2:a
[A]  2:b
[A] for obj in com.kabira.snippets.managedobjects.Range$MyObject
     using ByNumberDescription where (number > 1 && number < 5) { }
[A]  4:a
[A]  3:a
[A]  2:b
[A]  2:a
```

Atomic Create or Select

Often an application wants to atomically select a unique object if it exists, and if it does not exist, to create the instance. The query interface provides a mechanism to do this using the

`KeyQuery<T>.getOrCreateSingleResult` method. Query scope (see the section called "Query Scope" on page 70) can be used to select or create a unique instance on only the local node, a set of nodes in the cluster, or all nodes in the cluster. The `KeyQuery<T>.getOrCreateSingleResult` method always returns an object to the caller.

The `KeyQuery<T>.getOrCreateSingleResult` method works even when multiple threads may be selecting the same instance simultaneously. If multiple threads are attempting to perform a select for an object that does not exist, one of the threads will create the new instance, and all other threads will block until the transaction in which the object was created commits. The other threads are released after the transaction completes and the select will return the new instance.

The lock mode specified in the the `KeyQuery<T>.getOrCreateSingleResult` method is only used if the object already exists. It is ignored if the object was created. Newly created objects always have a transactional write lock.

See the section called "Constructors" on page 87 for details on constructor execution when an object is created by the `KeyQuery<T>.getOrCreateSingleResult` method.

Example 5.24 on page 83 demonstrates how to locate a unique instance on the local node, either by selecting or instantiating the object. It also shows how to determine whether the object was created or selected using the `Transaction.createdInTransaction(Object object)` and `Transaction.modifiedInTransaction(Object object)` methods.

Example 5.24. Atomic Create of Unique Keyed Object

```
//     $Revision: 1.1.2.4 $
package com.kabira.snippets.managedobjects;

import com.kabira.platform.KeyFieldValueList;
import com.kabira.platform.KeyManager;
import com.kabira.platform.KeyQuery;
import com.kabira.platform.LockMode;
import com.kabira.platform.Transaction;
import com.kabira.platform.ManagedObject;
import com.kabira.platform.QueryScope;
import com.kabira.platform.Transaction.Rollback;
import com.kabira.platform.annotation.Key;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.annotation.KeyField;

/**
 * Atomically creating a uniquely keyed object
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class AtomicCreate
{
    /**
     * A managed object with a unique key
     */
    @Managed
    @Key(name = "ByNumber",
    fields =
    {
        "number"
    },
    unique = true,
    ordered = false)
    private static class MyObject
```

```
{
    /**
     * Create MyObject and initialize description
     *
     * @param aNumber Initialize number field
     * @param aDescription Initialize description field
     */
    public MyObject(
        @KeyField(fieldName = "number") int aNumber,
        @KeyField(fieldName = "description") String aDescription)
    {
        this.number = aNumber;
        this.description = aDescription;
    }

    /**
     * Create MyObject with default description
     *
     * @param number Initialize number field
     */
    public MyObject(
        @KeyField(fieldName = "number") int number)
    {
        this.number = number;
        this.description = "default description";
    }
    final int number;
    final String description;
}

/**
 * Main entry point
 *
 * @param args Not used
 */
public static void main(String[] args)
{
    //
    //    Create an object
    //
    new Transaction("Create Object")
    {
        @Override
        protected void run() throws Rollback
        {
            //
            //  Create object with a key value of 1
            //
            MyObject myObject = new MyObject(1, "start of world");
            displayStatus("Create with constructor", myObject);
        }
    }.execute();

    new Transaction("Atomic Create")
    {
        @Override
        protected void run() throws Rollback
        {
            //
            //  Create required query objects
            //
            KeyManager<MyObject> keyManager = new KeyManager<MyObject>();
            KeyQuery<MyObject> keyQuery = keyManager.createKeyQuery(
                MyObject.class, "ByNumber");
            KeyFieldValueList keyFieldValueList = new KeyFieldValueList();
            KeyFieldValueList additionalFields = new KeyFieldValueList();
```

```
//
//    Set the query scope to local node only (default value)
//    This could also be set to cluster wide, or a sub-set of
//    the nodes in the cluster.
//
keyQuery.setQueryScope(QueryScope.QUERY_LOCAL);

//
//  Define the query to find object 1 which was created above
//
keyFieldValueList.add("number", 1);
keyQuery.defineQuery(keyFieldValueList);

//
//  Set up additionalFields
//
additionalFields.add("description", "create new object");

//
//  Atomically select or create a new object.  This will return
//  the original object created above since a key value of 1 is
//  being used.  The Transaction.createdInTransaction
//  and Transaction.modifiedInTransaction methods
//  can be used to determine whether the object was
//  created or selected.  Also note that the
//  description field is the value that was set when the object
//  was created.
//
//  This method always returns a result and never throws
//  an ObjectNotUniqueError exception.
//
System.out.println(keyQuery);
MyObject myObject = keyQuery.getOrCreateSingleResult(
    LockMode.WRITELOCK, additionalFields);
displayStatus("Original", myObject);

//
//    Delete the object
//
ManagedObject.delete(myObject);

//
//  Reset the query to use a value of 2
//
keyFieldValueList.clear();
keyFieldValueList.add("number", 2);
keyQuery.defineQuery(keyFieldValueList);

//
//  This will return a new object instance because no object
//  exists with a key value of 2.  The description field is
//  set to the value defined in additionalFields parameter.
//
//  The object is created using the constructor that passes
//  in description.
//
System.out.println(keyQuery);
myObject = keyQuery.getOrCreateSingleResult(
    LockMode.WRITELOCK, additionalFields);
displayStatus("New Key Value 2", myObject);
ManagedObject.delete(myObject);

//
//  Reset the query to use a value of 3
//
keyFieldValueList.clear();
keyFieldValueList.add("number", 3);
```

```
                keyQuery.defineQuery(keyFieldValueList);

                //
                //  This will return a new object instance because no object
                //  exists with a key value of 3.
                //
                //  The object is created using the constructor without a
                //  description parameter.
                //
                System.out.println(keyQuery);
                myObject = keyQuery.getOrCreateSingleResult(LockMode.WRITELOCK, null);
                displayStatus("New Key Value 3", myObject);
                ManagedObject.delete(myObject);
            }
        }.execute();
    }

    private static void displayStatus(final String label, final MyObject myObject)
    {
        System.out.println("==== " + label + " ===");
       System.out.println("Created    : " + Transaction.createdInTransaction(myObject));

       System.out.println("Modified   : " + Transaction.modifiedInTransaction(myObject));

        System.out.println("");
    }
}
```

When this snippet is run it outputs (annotation added):

```
#
#    Object created using new
#
[A] ==== Create with constructor ===
[A] Created    : true
[A] Modified   : true
[A] Deleted    : false
[A] Description: start of world

#
#    Select of original object, notice that
#    created is false and description is
#    initial value
#
[A] select obj from com.kabira.snippets.managedobjects.AtomicCreate$MyObject
    using ByNumber where (number == 1);
   Java constructor: public com.kabira.snippets.managedobjects.AtomicCreate$MyObject(int)
[A] ==== Original ===
[A] Created    : false
[A] Modified   : false
[A] Deleted    : false
[A] Description: start of world

#
#    Object is deleted.  Notice deleted is true
#
[A] ==== Deleted ===
[A] Created    : false
[A] Modified   : false
[A] Deleted    : true
[A]

#
#    Object created with getOrCreateSingleResult
#    using constructor with description parameter
#    Notice description value was set
#
```

```
[A] select obj from com.kabira.snippets.managedobjects.AtomicCreate$MyObject
    using ByNumber where (number == 2);
   Java constructor: public com.kabira.snippets.managedobjects.AtomicCreate$MyObject(int)
[A] ==== New Key Value 2 ===
[A] Created    : true
[A] Modified   : true
[A] Deleted    : false
[A] Description: create new object

#
#    Object created with getOrCreateSingleResult
#    using constructor without description parameter.
#    Notice default description value.
#
[A] select obj from com.kabira.snippets.managedobjects.AtomicCreate$MyObject
    using ByNumber where (number == 3);
   Java constructor: public com.kabira.snippets.managedobjects.AtomicCreate$MyObject(int)
[A] ==== New Key Value 3 ===
[A] Created    : true
[A] Modified   : true
[A] Deleted    : false
[A] Description: default description
```

Constructors

The `KeyQuery<T>.getOrCreateSingleResult` method may create a new object. To create the object a constructor is called on the object being created. Keyed objects have constructors that at a minimum must have parameters to initialize the key fields. The constructor may also take additional parameters for other application specific initialization.

The `@KeyField` annotation is used to map constructor parameters to object fields. If the `@KeyField` annotation is specified, it must be specified for all parameters in the constructor. If it is not specified for all parameters an exception is thrown at runtime.

```
[A] com.kabira.platform.KeyConstructor$IncompleteAnnotations: Constructor
      public sandbox.KeyConstructor(java.lang.Integer,java.lang.String) is missing one
of more KeyField annotations
[A] at com.kabira.platform.KeyConstructor.findAnnotatedConstructor(KeyConstructor.java:275)
[A] at com.kabira.platform.KeyConstructor.createObject(KeyConstructor.java:205)
[A] at com.kabira.platform.KeyConstructor.invokeConstructor(KeyConstructor.java:105)
[A] at com.kabira.platform.KeyQuery.getOrCreateSingleResult(KeyQuery.java:305)
[A] at sandbox.DoIt$1.run(KeyConstructor.java:53)
[A] at com.kabira.platform.Transaction.execute(Transaction.java:303)
[A] at sandbox.DoIt.main(KeyConstructor.java:36)
[A] at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
[A] at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
[A] at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
[A] at java.lang.reflect.Method.invoke(Method.java:597)
[A] at com.kabira.platform.MainWrapper.invokeMain(MainWrapper.java:49)
```

Here is the definition of the `@KeyField` annotation:

```
package com.kabira.platform.annotation;

import java.lang.annotation.*;

/** This annotation describes the mapping between constructor arguments
  * and the key field that the argument sets.
  */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.PARAMETER)
public @interface KeyField
{
    /** The field name this parameter maps to.
```

```
     */
    String fieldName();
}
```

The `@KeyField` annotation is not inherited. It must be specified on every constructor that will be called when the `KeyQuery<T>.getOrCreateSingleResult(...)` method is used to create objects. The `KeyQuery<T>.getOrCreateSingleResult(...)` only attempts to execute constructors on the actual object type being created. It does not search for, or excecute, any constructors inherited from parent types.

If the `@KeyField` annotation is not specified on a constructor for an object being created using `KeyQuery<T>.getOrCreateSingleResult`, no constructor is called. The key fields are correctly updated, any additional fields are initialized, and the object is created, but any behavior in the constructor is not executed.

See Example 5.24 on page 83 for an example of the `@KeyField` annotation.

Partitioned objects

The `KeyQuery<T>.getOrCreateSingleResult` method on a partitioned object, when the query scope is the local node, has different behavior depending on whether the object being selected is mapped to a partition definition on the local node (see Chapter 7 for details on partitioning) or not.

When the `KeyQuery<T>.getOrCreateSingleResult` method is called using a local node query scope, if the key does not exist on the local node, an instance is created with the key. If the instance is mapped to partition information on the local node, the create is dispatched to the active node for the partition. If the key value is found on the active node, the instance found on the active node is returned to the caller. This is a case where a local node query scope, actually performs a remote dispatch to ensure duplicate keys do not occur.

Executing the `KeyQuery<T>.getOrCreateSingleResult` method with a local node scope for partitioned objects on a node that does not have the partition definitions, will create an instance with a duplicate key. This can cause problems at a later time, if the partition containing the duplicate key is migrated to the local node. In general, executing the `KeyQuery<T>.getOrCreateSingleResult` method for partitioned objects should be avoided on nodes that do not contain the partition definition.

Inherited keys

When a key is inherited by a child type in a class hierarchy it is possible that a duplicate key exception will be reported by `KeyQuery<T>.getOrCreateSingleResult`. This is an application defect, since the same key value is being used for different child object types with a common shared parent key. Example 5.25 on page 88 demonstrates the problem.

Example 5.25. Duplicate keys with atomic create or select

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.managedobjects;

import com.kabira.platform.KeyFieldValueList;
import com.kabira.platform.KeyManager;
import com.kabira.platform.KeyQuery;
import com.kabira.platform.LockMode;
import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Key;
import com.kabira.platform.annotation.KeyField;
```

```
import com.kabira.platform.annotation.KeyList;
import com.kabira.platform.annotation.Managed;

/**
 * Duplicate keys with atomic selects
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class AtomicCreateDuplicateKey
{
    @Managed
    @KeyList(keys =
    {
        @Key(name = "ByName",
                unique = true,
                ordered = false,
                fields =
                {
                    "m_name"
                }
        )
    })
    private static class Parent
    {
        private final String m_name;

        Parent(@KeyField(fieldName = "m_name") String name)
        {
            this.m_name = name;
        }

        @Override
        public String toString()
        {
            return getClass().getSimpleName() + ": " + m_name;
        }
    }

    private static class ChildOne extends Parent
    {
        ChildOne(@KeyField(fieldName = "m_name") String name)
        {
            super(name);
        }
    }

    private static class ChildTwo extends Parent
    {
        ChildTwo(@KeyField(fieldName = "m_name") String name)
        {
            super(name);
        }
    }

    /**
     * Main entry point
     *
     * @param args Not used
     */
    public static void main(String [ ] args)
    {
        new Transaction("create objects")
        {
            @Override
```

```
            public void run()
            {
                //
                //  Create an instance using smith as the key
                //
                ChildOne c1 = new ChildOne("smith");
                System.out.println("INFO: " + c1);

                //
                //  Uniquely select on key smith, but use type ChildTwo
                //
                KeyManager<ChildTwo> km = new KeyManager<>();
                KeyQuery<ChildTwo> kq = km.createKeyQuery(ChildTwo.class, "ByName");
                KeyFieldValueList fvl = new KeyFieldValueList();

                fvl.add("m_name", "smith");
                kq.defineQuery(fvl);

                //
                //  Duplicate key will be thrown because no ChildTwo instance
                //  exists with key smith, but attempting to create instance
                //  causes a duplicate key with ChildOne
                //
                ChildTwo c2 = kq.getOrCreateSingleResult(LockMode.WRITELOCK, null);
                System.out.println("INFO: " + c2);
            }
        }.execute();
    }
}
```

When this snippet is run it outputs (with annotation added):

```
#
# ChildOne instance created
#
INFO: ChildOne: smith

#
# Duplicate key exception when attempting to create ChildTwo with same name
#
Java main class com.kabira.snippets.managedobjects.AtomicCreateDuplicateKey.main exited
 with an exception.
com.kabira.platform.ObjectNotUniqueError: Duplicate found for key
   'com.kabira.snippets.managedobjects.AtomicCreateDuplicateKey$Parent::ByName'
   in com.kabira.snippets.managedobjects.AtomicCreateDuplicateKey$ChildTwo,
   instance 3465071219:692692576:1421967247573317000:125.
   Duplicate is com.kabira.snippets.managedobjects.AtomicCreateDuplicateKey$ChildOne,
   instance 3465071219:2981644168:1421967247573317000:51. Key data:
[
    m_name = "smith"
]

    at com.kabira.platform.NativeRuntime.completeSMObject(Native Method)
    at
com.kabira.snippets.managedobjects.AtomicCreateDuplicateKey$ChildTwo.$_init_Impl(AtomicCreateDuplicateKey.java:65)

    at
com.kabira.snippets.managedobjects.AtomicCreateDuplicateKey$ChildTwo.<init>(AtomicCreateDuplicateKey.java:64)

    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at
sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:57)

    at
sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)

    at java.lang.reflect.Constructor.newInstance(Constructor.java:526)
```

```
    at com.kabira.platform.KeyConstructor.createObject(KeyConstructor.java:245)
    at com.kabira.platform.KeyConstructor.invokeConstructor(KeyConstructor.java:104)
    at com.kabira.platform.KeyQuery.getOrCreateSingleResult(KeyQuery.java:364)
    at
com.kabira.snippets.managedobjects.AtomicCreateDuplicateKey$1.run(AtomicCreateDuplicateKey.java:101)

    at com.kabira.platform.Transaction.execute(Transaction.java:484)
    at com.kabira.platform.Transaction.execute(Transaction.java:542)
    at
com.kabira.snippets.managedobjects.AtomicCreateDuplicateKey.main(AtomicCreateDuplicateKey.java:104)
```

Flushing objects

The `com.kabira.platform.CacheManager.CacheFlusher.flush()` method can be used to explicitly flush objects from shared memory. Support is provided for flushing:

- Local managed objects.
- Distributed objects.

Replica objects cannot be flushed since this would break the application redundancy guarantees. Attempting to flush a replica object does nothing.

When an object is flushed from shared memory all object and index data is removed.

Flushing a local object is equivalent to deleting the object. This implies that any delete triggers (see the section called "Triggers" on page 58) will also be called.

Applications that flush objects, must be designed to ensure that the object data is consistently available whether it was flushed or not. This can be accomplished using distributed queries and ensuring that objects are fetched from secondary stores as required.

A flush notifier can be installed to control whether or not an object should be flushed. Flush notifiers extend `com.kabira.platform.flusher.Notifier<T>` and implement the abstract `isFlushable` method. They are installed with the `com.kabira.platform.flusher.FlushManager.setNotifier(...)` method. Returning `true` from the `isFlushable` method allows the flush to complete. Returning `false` from the `isFlushable` method cancels the object flush. If a flush notifier is not installed, objects are always flushed.

Objects can be flushed from a flush notifier. Flushing an object in a flush notifier does not cause any installed flush notifiers to be called.

The Example 5.26 on page 91 snippet shows local objects being flushed with one of the flush attempts being cancelled by a flush notifier.

Example 5.26. Object flushing

```
//     $Revision: 1.1.2.2 $
package com.kabira.snippets.managedobjects;

import com.kabira.platform.CacheManager;
import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.flusher.Notifier;

/**
 * Flushing objects from shared memory
 * <p>
```

```
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class ObjectFlushing
{
    @Managed
    private static class A
    {
        A(final String name)
        {
            m_name = name;
        }

        String getName()
        {
            return m_name;
        }
        private final String m_name;
    };

    private static class FlushNotifier extends Notifier<A>
    {
        FlushNotifier(Class<A> a)
        {
            super(a);
        }

        @Override
        public boolean isFlushable(A a)
        {
            boolean flushable = a.getName().equals(_DO_NOT_FLUSH) != true;
            System.out.println("INFO: Notifier called for " + a.getName() + ":" +
flushable);
            return flushable;
        }

    }

    /**
     * Main entry point
     *
     * @param args not used
     */
    public static void main(String[] args)
    {
        //
        //    Install a flush notifier and create a local object
        //
        new Transaction("Initialize")
        {
            @Override
            protected void run() throws Rollback
            {
                CacheManager.CacheFlusher.setNotifier(new FlushNotifier(A.class));

                new A(_DO_NOT_FLUSH);
                new A("Flush away!");
            }
        }.execute();

        //
        //    Flush the objects
        //
        new Transaction("Flush Objects")
        {
```

```
            @Override
            protected void run() throws Transaction.Rollback
            {
                for (A a : ManagedObject.extent(A.class))
                {
                    System.out.println("INFO: Flushing " + a.getName());
                    CacheManager.CacheFlusher.flush(a);
                }
            }
        }.execute();

        //
        //    Report what is still in shared memory
        //
        new Transaction("Report")
        {
            @Override
            protected void run() throws Transaction.Rollback
            {
                for (A a : ManagedObject.extent(A.class))
                {
                    System.out.println("INFO: " + a.getName());
                }
            }
        }.execute();
    }

    static final String _DO_NOT_FLUSH = "Do not flush";
}
```

When this snippet is run it outputs (annotations added):

```
    #
    #    Flush object
    #
    INFO: Flushing Do not flush

    #
    #    Notifier called - cancel flush by returning false
    #
    INFO: Notifier called for Do not flush:false

    #
    #    Flush object
    #
    INFO: Flushing Flush away!

    #
    #    Notifier called - allow flush by returning true
    #
    INFO: Notifier called for Flush away!:true

    #
    #    Only the Do not flush object is left in shared memory
    #
    INFO: Do not flush
```

Named caches

Named caches provide a mechanism to control the caching policy of managed objects. Named caches can be created using either the administrative tools or an API. Named caches can be dynamically created and destroyed while an application is running. When managed object classes are associated with the named cache the caching policies defined by the named cache are applied.

The Example 5.27 on page 94 creates a zero-byte named cache and associates a managed object class with the cache. When the cache is flushed by the background flusher thread, the created object is deleted, the delete trigger is called, and the object is removed from shared memory.

Example 5.27. Named cache creation

```
//     $Revision: 1.1.2.2 $
package com.kabira.snippets.managedobjects;

import com.kabira.platform.CacheManager;
import com.kabira.platform.CacheManager.Cache;
import com.kabira.platform.DeleteTrigger;
import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;

/**
 * Defining a named cache
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class NamedCache
{
    @Managed
    private static class A implements DeleteTrigger
    {
        @Override
        public void uponDelete()
        {
            System.out.println("INFO: Object deleted");
            m_deleted = true;
        }
    };

    /**
     * Main
     *
     * @param args Arguments - none supported
     * @throws InterruptedException
     */
    public static void main(final String[] args) throws InterruptedException
    {
        createNamedCache();

        //
        //  Create an object in the just created named cache
        //
        new Transaction()
        {
            @Override
            protected void run() throws Transaction.Rollback
            {
                new A();
            }
        }.execute();

        //
        //    Wait for the flusher to delete the local
        //    object when it is flushed from the cache
        //
        while (m_deleted == false)
        {
            Thread.sleep(1000);
```

```
        }
    }

    //
    //  Create a zero byte named cached.
    //  Objects in the cache will be flushed
    //    by the back-ground flusher thread.
    //
    private static void createNamedCache()
    {
        new Transaction()
        {
            @Override
            protected void run() throws Transaction.Rollback
            {
                Cache cache = CacheManager.getOrCreateCache(A.class.getSimpleName());

                cache.setSizeBytes(0);
                cache.addClass(A.class);
            }
        }.execute();
    }

    private static boolean m_deleted = false;
}
```

The snippet outputs these messages when it is run:

```
INFO: Object deleted
```

The created cache can be displayed using the administrator command line tool after the snippet is executed:

```
administrator servicename=A display cache
Name = A
Objects In Cache = 0
Cache Size = No Caching
Cache Utilization = 0.0% (0/512.0M)
Shared Memory Utilization = 0.0% (0/512.0M)
Types In Cache = com.kabira.snippets.managedobjects.NamedCache$A
Flusher Sleep Interval = 1
Maximum Objects Per Flush = 0
```

Notice that caching has been disabled in the named cache because the cache size was set to zero, and there are no objects in the cache after the snippet completes because the object was flushed from shared memory.

Asynchronous methods

An asynchronous method is specified using the `@Asynchronous` annotation. The `@Asynchronous` annotation is inherited by overridden methods in child types. The isolation level of the transaction in which the asynchronous method is executed can be specified in the `@Asynchronous` annotation.

Asynchronous methods have these restrictions:

- asynchronous methods are only supported on managed objects.
- an asynchronous method must be declared as `void`.
- an asynchronous method cannot be declared `static`.

- asynchronous method parameters have the same restrictions as distributed method parameters. See the section called “Distributed method signature restrictions” on page 50.

Adding the `@Asynchronous` annotation to a method that does not meet the above restrictions will fail at class load time with a `java.lang.NoclassDefFoundError` exception.

```
Class com/kabira/snippets/managedobjects/AsynchronousMethod$MyObject failed class audit:

        [Asynchronous method queueWork must return void]
   Java main class com.kabira.snippets.managedobjects.AsynchronousMethod.main exited with
 an exception.
    java.lang.NoClassDefFoundError:
com/kabira/snippets/managedobjects/AsynchronousMethod$MyObject
     at
com.kabira.snippets.managedobjects.AsynchronousMethod$1.run(AsynchronousMethod.java:50)
     at com.kabira.platform.Transaction.execute(Transaction.java:303)
     at
com.kabira.snippets.managedobjects.AsynchronousMethod.main(AsynchronousMethod.java:41)
     at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
     at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
     at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
     at java.lang.reflect.Method.invoke(Method.java:597)
     at com.kabira.platform.MainWrapper.invokeMain(MainWrapper.java:49)
```

Example 5.28 on page 96 shows a snippet of declaring and using an asynchronous method.

Example 5.28. Asynchronous method

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.managedobjects;

import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.annotation.Asynchronous;

/**
 *  Asynchronous methods
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class AsynchronousMethod
{
    /**
     * A managed object
     */
    @Managed
    public static class MyObject
    {
        @Asynchronous
        void queueWork()
        {
            System.out.println(
                "Executed in transaction: "
                + Transaction.getIdentifier().toString());
        }
    };

    /**
     * Main entry point
     * @param args  Not used
```

```
     */
    public static void main(String [] args)
    {
        new Transaction("Asynchronous Method")
        {
            @Override
            protected void run() throws Rollback
            {
                System.out.println(
                    "Calling transaction: "
                    + Transaction.getIdentifier().toString());

                MyObject    e = new MyObject();
                e.queueWork();
           }
        }.execute();
    }
}
```

When this snippet is run it outputs the following:

```
[A] Calling transaction: 872:1
[A] Executed in transaction: 850:1
```

Queued asynchronous methods are discarded with a log message if the target object was destroyed before the method could be executed. Object lifecycle must be carefully managed to ensure that all queued asynchronous methods are executed before destroying objects.

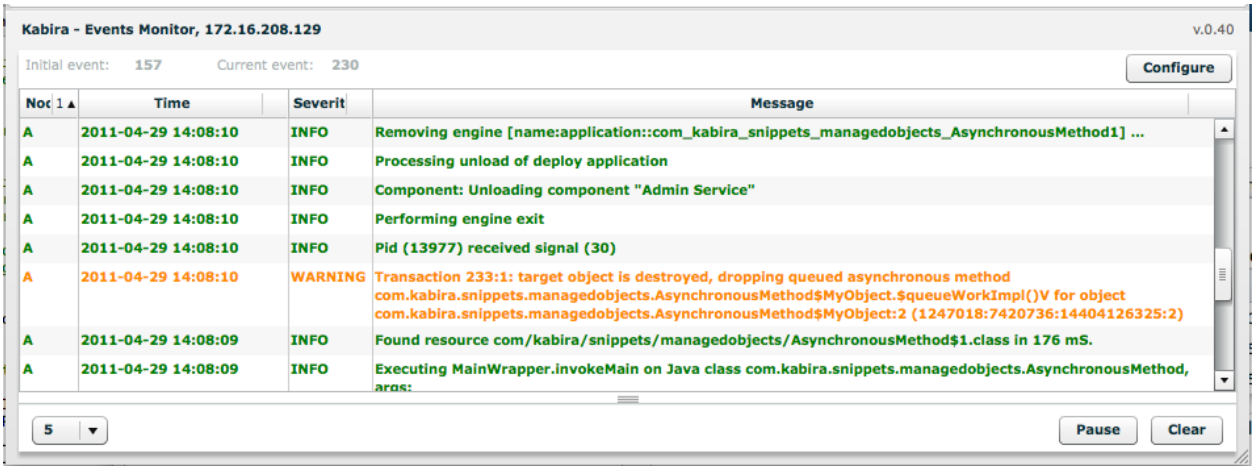

Figure 5.2. Destroyed object warning

Array copy-in/copy-out

When a field in a Managed Object is accessed, transactional locking and logging occur. This is true for primitive types, arrays, and objects.

Arrays can also be copied into a local array variable to avoid transactional locking or logging. This copy occurs implicitly if an array is passed into a method for execution. Shared memory backing an array is only modified when elements in the array are modified using the object reference containing the array. These cases are shown in the snippet below.

Array copies are a useful performance optimization when a large number of elements in an array are being modified in a single transaction.

Example 5.29. Array copy-in/copy-out

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.managedobjects;

import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;

/**
 *  Array copy-in/copy-out
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class Array extends Transaction
{
    /**
     * A managed object
     */
    @Managed
    public static class MyObject
    {
        int        value;
    }

    /**
     * A managed object containing other managed objects
     */
    @Managed
    public static class MyObjectContainer
    {
        /**
         * Create a new object
         */
        public MyObjectContainer()
        {
            super();

            sharedMemoryArray = new int[10];
            int        i;
            for (i = 0; i < 10; i++)
            {
                sharedMemoryArray[i] = i;
            }

            objectArray = new MyObject[2];
            for (i = 0; i < 2; i++)
            {
                MyObject    f2 = new MyObject();
                f2.value = i;
                objectArray[i] = f2;
            }
        }
        int         []    sharedMemoryArray;
        MyObject    []    objectArray;
    }
    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String [] args)
    {
        Array    array = new Array();
        array.execute();
```

```
    }

    /**
     * Transaction run method
     *
     */
    @Override
    protected void run()
    {
        MyObjectContainer    f = new MyObjectContainer();

        //
        //    Read lock f and make of copy of sharedMemoryArray in localArray
        //
        int    localArray[] = f.sharedMemoryArray;

        //
        //    This does not modify shared memory
        //
        localArray[2] = 6;

        System.out.println("localArray: " + localArray[2]
                           + " sharedMemoryArray: " + f.sharedMemoryArray[2]);

        //
        //    This modifies shared memory and takes a write lock on f
        //
        f.sharedMemoryArray[2] = 7;

        System.out.println("localArray: " + localArray[2]
                           + " sharedMemoryArray: " + f.sharedMemoryArray[2]);


        //
        //    This does not change the value of sharedMemoryArray in
//    shared memory
        //
        modifyIntArray(localArray);

        System.out.println("localArray: " + localArray[0]
                           + " sharedMemoryArray: " + f.sharedMemoryArray[0]);


        //
        //    This does not modify the value of sharedMemoryArray in shared
        //    memory.  It takes a read lock on f.
        //
        modifyIntArray(f.sharedMemoryArray);

        System.out.println("localArray: " + localArray[0]
                           + " sharedMemoryArray: " + f.sharedMemoryArray[0]);


        //
        //    This copies the value of localArray into shared memory
        //    and takes a write lock on f.
        //
        f.sharedMemoryArray = localArray;

        System.out.println("localArray: " + localArray[0]
                           + " sharedMemoryArray: " + f.sharedMemoryArray[0]);

        //
        //    This copies only the object references in objectArray to a local
        //    array
        //
        MyObject    localF2Array[] = f.objectArray;
```

```
        //
        //    This updates shared memory through the object reference copied
        //    into the local array
        //
        localF2Array[0].value = 8;
        System.out.println("f2.value: " + f.objectArray[0].value);
    }

    /**
     * Modify the array passed as a parameter
     * @param arg   Array to modify
     */
    public void modifyIntArray(int [] arg)
    {
        arg[0] = 5;
    }
}
```

When Example 5.29 on page 98 is run it outputs (annotation added):

Example 5.30. Array copy-in/copy-out output

```
#
#     Modify local array with a value of 6
#
[A] localArray: 6 sharedMemoryArray: 2

#
#     Modify shared memory with a value of 7
#
[A] localArray: 6 sharedMemoryArray: 7

#
#     Modify local array with a value of 5
#
[A] localArray: 5 sharedMemoryArray: 0

#
#    Modify shared memory array passed to a method with a value of 5
#
[A] localArray: 5 sharedMemoryArray: 0

#
#     Copy local array into shared memory array
#
[A] localArray: 5 sharedMemoryArray: 5

#
#     Modify shared memory using an object reference in a local array
#
[A] f2.value: 8
```

Reflection limitations

Managed Objects support `java.lang.reflect` except for modification of an array element. Array element modifications are not updated in shared memory. Example 5.31 on page 100 shows this behavior.

Example 5.31. Reflection behavior

```
//     $Revision: 1.1.2.1 $
```

```
package com.kabira.snippets.managedobjects;

import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;
import java.lang.reflect.Array;

/**
 *  This snippet is used to demonstrate reflection limitations
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class Reflection
{
    /**
     * A managed object
     */
    @Managed
    public static class MyObject
    {
        MyObject()
        {
            super();

            intArray = new int [10];
            int        i;

            for (i = 0; i < 10; i++)
            {
                intArray[i] = i;
            }
        }
        int    []    intArray;
    }
    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String [] args)
    {
        new Transaction("Reflection")
        {
            @Override
            protected void run() throws Rollback
            {
                 MyObject    i = new MyObject();

                //
                //    Read the 5th element
                //
                System.out.println("intArray[5] == " + Array.getInt(i.intArray, 5));

                //
                //    Modify the 5th element
                //
                Array.setInt(i.intArray, 5, 0);


                //
                //    Read the 5th element - still 5
                //
                System.out.println("intArray[5] == " + Array.getInt(i.intArray, 5));
           }
        }.execute();
```

```
    }
}
```

When run Example 5.31 on page 100 outputs (annotation added):

Example 5.32. Reflection behavior output

```
#
#    Original value of 5th element of intArray
#
[A] intArray[5] == 5

#
#    5th element still contains a value of 5 even after being set to 0 by Reflection API
#
[A] intArray[5] == 5
```

6

Distributed computing

This chapter describes how to add distributed computing features to your application. All managed objects are distributed objects by default. No additional annotation, API calls, or configuration, is required.

This chapter briefly describes the behavior of distributed objects. Highly available objects, which are also distributed objects are described in Chapter 7.

Distributed object life cycle

Distributed objects have the same life cycle as a non-distributed managed object. See the section called “Managed object life cycle” on page 51 for details.

However, if an object maintains a reference to a distributed object, that object can be deleted on a remote node and the local object now has a stale reference. This stale reference is not detected until a field is accessed or modified, or a method is invoked. When this happens a `java.lang.NullPointerException` is thrown to the application.

```
class A { }

class B
{
   A a // This can become stale if an instance of B is a distributed object
}
```

The `java.lang.NullPointerException` can be avoided by ensuring that a write lock is taken on distributed object references before a field or method is accessed. The recommended way to take a write lock is to select the object, if it has an index, and specify a lock type of write. This will force a distributed lock request to the remote node which will either return the instance write locked, or an empty reference if the object was deleted.

In general, it is recommended that applications coordinate distributed object deletes to avoid any special case exception handling or locking code on remote nodes.

Constructors

Constructors are always executed on the node where `new` is called. This is true even if the master node for the object will be a remote node, for example a partitioned object whose active node is not the local node. Figure 6.1 shows `new` being called for an `Order` object on `Node One`. The constructor executes on `Node One` and then the object is created on `Node Two`. When the creation on `Node Two` completes, `new` returns to the caller.

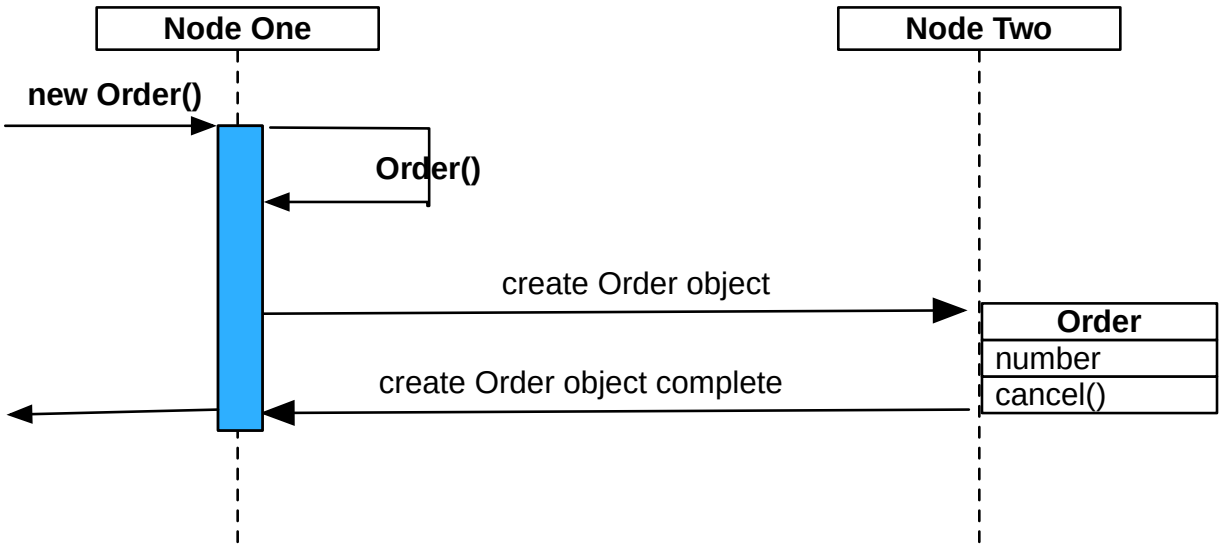

Figure 6.1. Constructor execution

Unavailable node exceptions

Attempting to access a distributed object from a remote node when the master node for the object is down will cause a `com.kabira.platform.ResourceUnavailableException` exception to be thrown.

A remote node is detected to be down when the following action is attempted on a distributed reference:

- method invocation
- object deletion
- field modification
- field access when the object data is not available on the local node

Highly available objects that have replica nodes defined will never throw the `com.kabira.platform.ResourceUnavailableException` exception because the master node of the distributed object fails over to a replica node when the current active node for the partition fails. This failover is transparent to applications (see the **TIBCO ActiveSpaces® Transactions Architect's Guide**).

Remote objects

A remote object is obtained in one of the following ways:

- An object pushed to a remote node as part of replication. This mechanism is described in the Chapter 7.
- A query pulls remote objects to the local node. This mechanism is described in the section called "Queries" on page 69.
- An object returned from a method invocation on a remote object.
- An external directory service. This approach is not discussed in this document.

When a remote object is obtained on a node, all fields for the object are read to the node.

For non-replicated distributed objects, the field data is refreshed when:

- a method is executed on a remote object that modifies the object. All field data is read to the node when the method completes. If a method does not modify the object, the field data is not refreshed.
- a state conflict is detected (see the section called "State conflict" on page 109).
- the object is explicitly flushed (see the section called "Flushing objects" on page 91). The field data is refreshed the next time the object is accessed.

Replicated objects are automatically refreshed as they are modified on their active node.

Replicated objects are only refreshed if the local node is a replica in the partition definition. If the local node is not in the partition definition, this object behaves as a non-replicated distributed object.

The default caching policy for non-replicated distributed objects is cache never, which means that objects are only cached on the local node for the duration of the transaction. The object is flushed when the transaction commits. See the section called "Flushing objects" on page 91 for more details.

The Example 6.1 on page 105 demonstrates using a distributed query to access remote objects. The snippet performs the following steps:

1. Creates a named cache to contain the objects being created.
2. Creates an object on each node in a cluster.
3. Uses a distributed cardinality query to wait for all nodes to create objects.
4. Executes a distributed extent query to cache all objects in the local node.
5. Displays the contents of the local extent, showing all of the cached objects.
6. Waits for an explicit shutdown.

Example 6.1. Accessing remote objects

```
//     $Revision: 1.1.2.6 $
package com.kabira.snippets.distributedcomputing;

import com.kabira.platform.CacheManager;
```

```
import com.kabira.platform.LockMode;
import com.kabira.platform.ManagedObject;
import com.kabira.platform.QueryScope;
import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.property.Status;

/**
 * This snippet demonstrates how to access distributed objects
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainname</b> = Development
 * </ul>
 */
public class RemoteReferences
{
    @Managed
    private static class Distributed
    {
        Distributed()
        {
            m_message = "Created on " + _NODE_NAME;
        }

        String getMessage()
        {
            return m_message;
        }

        String getCreatedOnNode()
        {
            return _NODE_NAME;
        }

        private final String m_message;
        private final static String _NODE_NAME = System.getProperty(Status.NODE_NAME);
    }

    /**
     * Main entry point
     *
     * @param args Not used
     * @throws InterruptedException Sleep interrupted
     */
    public static void main(String[] args) throws InterruptedException
    {
        //
        //  Initialize the snippet
        //
        initialize();

        //
        //  Wait for all nodes to create objects
        //
        waitForInitialization();

        //
        //  Cache all distributed objects on local node
        //
        cacheDistributedObjects();

        //
        //  Display distributed objects
        //
        displayDistributedObjects();
```

```
        //
        //  Wait for shutdown
        //
        waitForStop();
    }

    //
    //  Display the distributed objects
    //
    private static void displayDistributedObjects()
    {
        new Transaction("Display Distributed Objects")
        {
            @Override
            protected void run()
            {
                String label;

                //
                //  Access objects using local extent
                //  This returns only the objects located on the loca node
                //
                System.out.println("Found in local extent on " + _NODE_NAME);
                for (Distributed d : ManagedObject.extent(Distributed.class))
                {
                    System.out.println("\t" + d.getMessage());
                }
            }
        }.execute();
    }

    //
    // Initialize the snippet
    //
    private static void initialize()
    {
        new Transaction("Initialization")
        {
            @Override
            protected void run() throws Rollback
            {
                //
                // Set up a cache always distributed object cache on all
                // nodes.  This overrides the default distributed object
                // behavior which flushes distributed objects after they
                // are accessed.
                //
                CacheManager.Cache cache =
                    CacheManager.getOrCreateCache("Remote References");
                cache.setSizePercent(100);
                cache.addClass(Distributed.class);
                cache.enable();

                new Distributed();
            }
        }.execute();
    }

    //
    // Wait for objects to be created on all nodes
    //
    private static void waitForInitialization() throws InterruptedException
    {
        while (m_count != 3)
        {
            Thread.sleep(5000);
```

```
            new Transaction("Wait for Creation")
            {
                @Override
                protected void run()
                {
                    //
                    //  Execute cluster-wide cardinality to determine number
                    //  of objects created
                    //
                    m_count = ManagedObject.cardinality(
                    Distributed.class,
                    QueryScope.QUERY_CLUSTER);
                }
            }.execute();
        }
    }

    //
    //  Perform a cluster-wide extent query to cache objects on local node
    //
    private static void cacheDistributedObjects() throws InterruptedException
    {
        new Transaction("Cache Objects")
        {
            @Override
            protected void run()
            {
                //
                // Execute cluster wide extent query to pull all
                // objects to the local node
                //
                for (Distributed d : ManagedObject.extent(
                Distributed.class,
                QueryScope.QUERY_CLUSTER,
                LockMode.READLOCK))
                {
                    System.out.println("Cached object from " + d.getCreatedOnNode());
                }
            }
        }.execute();
    }

    //
    //  Wait for termination
    //
    private static void waitForStop() throws InterruptedException
    {
        System.out.println("Waiting for stop...");

        while (true)
        {
            Thread.sleep(1000);
        }
    }
    private final static String _NODE_NAME = System.getProperty(Status.NODE_NAME);
    private static int m_count = 0;
}
```

When this snippet is run it outputs (annotation added):

```
#
#  Cache all objects on node C by doing a distributed extent query
#
[C] Cached object from C
[C] Cached object from A
[C] Cached object from B
```

```
#
#  Shows all cached objects in local extent on node C
#
[C] Found in local extent on C
[C]  Created on C
[C]  Created on A
[C]  Created on B
[C] Waiting for stop...

#
#  Cache all objects on node B by doing a distributed extent query
#
[B] Cached object from C
[B] Cached object from A
[B] Cached object from B

#
#  Shows all cached objects in local extent on node B
#
[B] Found in local extent on B
[B]  Created on C
[B]  Created on A
[B]  Created on B
[B] Waiting for stop...

#
#  Cache all objects on node A by doing a distributed extent query
#
[A] Cached object from C
[A] Cached object from A
[A] Cached object from B
[A] Found in local extent on A

#
#  Shows all cached objects in local extent on node A
#
[A]  Created on C
[A]  Created on A
[A]  Created on B
[A] Waiting for stop...
```

State conflict

A state conflict is reported by TIBCO ActiveSpaces® Transactions when a write operation from a remote node detects that the data on the local node has changed underneath it. This is possible in a distributed system because an object may be modified from multiple nodes in the system. The `com.kabira.platform.StateConflictError` exception is thrown when a state conflict is detected.

This exception should never be caught by the application. It is used by TIBCO ActiveSpaces® Transactions to managed state conflicts transparently. If a state conflict is detected an exception is returned to the remote node. The transaction is then rolled back and replayed. This causes all object field data to be refreshed on the remote node. The application is never aware that a state conflict occurred.

Extents

Global extents are not automatically maintained for distributed objects. As remote objects are accessed on a node, they are added to the extent on the node. Highly available replicated objects maintain distributed extents since they are automatically pushed to multiple nodes in a cluster.

Guidelines

Here are some general guidelines for distributed programming with TIBCO ActiveSpaces® Transactions .

- All modifications to a distributed object should be done on one node. This reduces the chance of state conflicts which cause performance degradation. The best way to enforce this is to use methods to perform field updates. The method will execute on the master node transparently.
- Eliminate distributed deadlocks from an application. Distributed deadlock detection uses a timeout to detect a deadlock. This implies that a distributed transaction will wait the entire value of the timeout value before a deadlock is reported. During this period of time the transaction is stalled.
- A high availability partition with a single node in the node list can be used to create an object instance on a specific node.
- Distributed queries should be used to discover remote references for nodes in a cluster.
- Evaluate which nodes the application classes must be installed on. A classs must be installed on all nodes that create or host distributed objects.

7

High availability

This chapter describes how to use the high availability services.

Defining a partitioned object

A partitioned object is a managed object with a *partition mapper* installed. An object is highly available if at least one replica node is defined for the partition. There is no difference in the behavior of Managed Objects when they have a *partition mapper* installed.

To add partitioned objects to an application, the following is required:

- Define a *partition mapper* and install it on all managed object classes that should be partitioned.
- Define and enable the required partitions.

Partitioned object life cycle

Partitioned objects have the same life cycle as a non-partitioned managed object. See the section called “Managed object life cycle” on page 51 for details. When a partitioned object is deleted any replica copies of the object are also deleted. See the section called “Distributed object life cycle” on page 103 other details on distributed object life-cycle.

Restrictions

A partitioned managed object adds this additional restriction to a managed object definition (the section called “Restrictions” on page 49):

- Any parent classes with keys must be abstract.

This restriction is audited when the *Partition Mapper* is installed (see Example 7.1 on page 113). If the audit fails, a `com.kabira.platform.ManagedClassError` is thrown by the

com.kabira.platform.highavailability.PartitionManager.setMapper(...) method.

Defining a partition mapper

Partition mappers are defined by extending com.kabira.platform.highavailability.PartitionMapper and implementing the getPartition method. Partition mappers are installed on a specific class using the PartitionManager.setMapper() method.

All child classes extending a parent class inherit the parent's partition mapper. The parent's partition mapper can be overridden in a child class by installing a partition mapper on the child. Only a child's partition mapper is called when the child is created.

The getPartition method is called when objects are created, or re-partitioning (the section called "Updating partition mapping" on page 126) is requested. The method is passed an object of the type for which it was registered. The getPartition method must return a valid partition name for the object. The object is assigned to the returned partition name.

Partition mappers must be installed on each node in the cluster that has the managed object type installed. This includes all replica nodes for the managed object type. Failure to do so will cause inconsistent behavior in the cluster because the type will be partitioned on some nodes, but not on others. The partition mapper should be installed as part of application initialization before a node joins the cluster and before any objects of that type are created.

When a partition mapper is installed an audit is performed to validate the current state of partitioned objects on the node. The audit to perform can be one of:

- IGNORE_PARTITIONING - do not perform any validation of previously created object instances for the type on which the partition mapper is being installed. Any objects created before the partition mapper is installed will not be partitioned.
- VERIFY_DEFER_INSTALL - verify that there are no previously created object instances for the type on which the partition mapper is being installed. If there are object instances a com.kabira.platform.ResourceUnavailableException is thrown. If the type has not loaded, the installation of the partition mapper is deferred until the type is loaded.
- VERIFY_PARTIONING - verify that there are no previously created object instances for the type on which the mapper is being installed. If there are object instances a com.kabira.platform.ResourceUnavailableException is thrown.

The decision on which partition should be associated with an object is based on an application specific criteria. For example all customers on the west coast could be in a partition named westcoast, while all customers on the east coast could be in a partition named eastcoast. Partitions could also be assigned based on load balancing, or other system resource criteria.

If an invalid partition name is returned from the getPartition method a com.kabira.platform.ResourceUnavailableException is thrown.

Partition mappers can be cleared using the PartitionManager.clearMapper() method.

Example 7.1 on page 113 shows the definition and installation of a partition mapper.

Example 7.1. Defining , installing, and clearing a partition mapper

```
//     $Revision: 1.1.2.4 $
package com.kabira.snippets.highavailability;

import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.highavailability.PartitionManager;
import com.kabira.platform.highavailability.PartitionMapper;

/**
 * Defining and installing a partition mapper.
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A</li>
 * </ul>
 * </p>
 */
public class HighAvailability
{
    @Managed
    private static class MyObject
    {
    };

    //
    //  Partition mapper that just returns a hard-coded partition name
    //
    private static class MyPartitionMapper extends PartitionMapper
    {
        @Override
        public String getPartition(Object obj)
        {
            return "Unknown Partition";
        }
    }

    /**
     * Main entry point
     *
     * @param args Not used
     */
    public static void main(String[] args)
    {
        new Transaction("High Availability")
        {
            @Override
            protected void run() throws Rollback
            {
                //
                //    Define partition mapper properties to audit for any
                //    unpartitioned objects
                //
               PartitionMapper.Properties properties = new PartitionMapper.Properties();

                properties.setAudit(PartitionMapper.Properties.Audit.VERIFY_PARTIONING);


                //
                //  Install the partition mapper
                //
                PartitionManager.setMapper(
                    MyObject.class, new MyPartitionMapper(), properties);

                //
                //  Create an instance of MyObject.
```

```
                //    This will fail because partition mapper
                //    is returning an unknown partition
                //
                try
                {
                    new MyObject();
                }
                finally
                {
                    //
                    //    Clear the partition mapper
                    //
                    PartitionManager.clearMapper(MyPartitionMapper.class);
                }
            }
        }.execute();
    }
}
```

When Example 7.1 on page 113 is run it fails with the following output:

```
[A] Java main class com.kabira.snippets.highavailability.HighAvailability.main exited
with an exception.
[A] com.kabira.platform.ResourceUnavailableException:
          could not find partition 'Unknown Partition' for object
         'com.kabira.snippets.highavailability.MyObject:1 (1352756:1794744:14616904711:1
 offset 68152136)'
[A]  at com.kabira.platform.ManagedObject.createSMObject(Native Method)
[A]  at com.kabira.snippets.highavailability.MyObject.<init>(HighAvailability.java:11)
[A]  at
com.kabira.snippets.highavailability.HighAvailability$1.run(HighAvailability.java:56)
[A]  at com.kabira.platform.Transaction.execute(Transaction.java:303)
[A]  at
com.kabira.snippets.highavailability.HighAvailability.main(HighAvailability.java:43)
[A]  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
[A]  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
[A]  at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
[A]  at java.lang.reflect.Method.invoke(Method.java:597)
[A]  at com.kabira.platform.MainWrapper.invokeMain(MainWrapper.java:49)
```

This is expected, because the partition name returned by `getPartition` was not defined. Defining partitions is described in the section called “Defining and enabling partitions” on page 114.

Defining and enabling partitions

There are two mechanisms to define partitions:

- Programmatically using an API
- Administration tools

Programmatic definition of partitions is covered in this section. See the **TIBCO ActiveSpaces® Transactions Administration Guide** for details on defining partitions using the administration tools.

Mixing partition management via the API and administration commands must be done carefully to ensure that an operator using the administrative commands does not see unexpected results.

Partitions can be dynamically defined as needed. A partition definition consists of:

- a cluster-wide unique name
- optional partition properties
- initial active node
- optional list of replica nodes ordered by priority

The optional replica node list is used to define the replica nodes for a partition. The replica nodes are specified in priority order. If the current active node for a partition fails, the first replica node becomes the active node. Each replica node definition consists of:

- node name
- whether to use synchronous or asynchronous replication

Once a partition is defined it cannot be removed.

Partitions should be defined and enabled on all nodes in the cluster that need knowledge of the partition.

The supported partition properties are:

- **broadcast partition definition updates** - controls the broadcasting of partition definition updates to all nodes in the cluster. Disabling this behavior is useful for simulating a multi-master scenario. See the section called “Simulating a multi-master scenario” on page 153 for details.
- **force replication** - used to control replication during a migration. If this value is set to `true` replication will be done to all currently active replica nodes for the partition. The default value for this property causes no replication to be done to the current replica nodes during a migration. This option can be used to force resynchronization of all replica nodes for a partition. In general, this should be avoided and a replica node should be brought offline and back online to resynchronize replica data.
- **number of threads** - controls the number of threads that are used to perform object migration. If the number of objects in the partition is less than the value of the objects locked per transaction property, only a single thread will be used. If the value of the number of objects locked per transaction property is zero, or the calling transaction has one or more partitioned objects locked, the value of this property is ignored and the caller's thread will be used to perform the migration.
- **objects locked per transaction** - controls the number of objects that are locked during a migration or update per transaction. Setting this property to a value greater than zero allows application work to continue concurrently while a partition is being migrated or updated. Setting this property to a value of zero causes all objects in the partition to be locked in a single transaction.
- **restore from node** - define which node a partition should be restored from in a multi-master scenario. This property should be used if the partition is active on multiple nodes in the cluster. If this property is set, the current node must be the current active node for the partition when the partition is defined. If this value is not specified a cluster wide broadcast is used to determine which node a restore should be done from.
- **sparse partition audit** - controls whether the node list defined for a sparse partition is audited to match the node list of the current active node when the sparse partition is enabled. Disabling this audit may be useful to resolve partition definition ordering dependencies across nodes in a cluster. For example, a sparse partition may be restored before the partition on an active node is

restored following a failover. In this case, if the sparse partition audit is not disabled, the enabling of the sparse partition will fail with a node list mismatch exception.

Example 7.2 on page 118 shows the steps required to define and enable a new partition. The steps are:

1. Call `PartitionManager.definePartition(...)` to define the partition name, the optional partition properties, the initial active node, and optionally one or more replica nodes.

2. Call `PartitionManager.enablePartitions(...)` to enable all defined partitions, or `partition.enable(...)` to enable a specific partition, to the cluster.

The `PartitionManager.definePartition(...)` method is synchronous - it blocks until the active node in the partition definition is available. If the active node for the partition is not available, a `com.kabira.platform.ResourceUnavailableException` is thrown, for example:

```
Java main class sandbox.highavailability.InvalidActiveNode.main exited with an exception.
com.kabira.platform.ResourceUnavailableException: Remote node 'bogus' cannot be accessed,
 current state is 'Undiscovered'
 at com.kabira.platform.disteng.DEPartitionManager.definePartition(Native Method)
 at
com.kabira.platform.highavailability.PartitionManager.definePartition(PartitionManager.java:810)
 at
com.kabira.platform.highavailability.PartitionManager.definePartition(PartitionManager.java:810)

 at
com.kabira.platform.highavailability.PartitionManager.definePartition(PartitionManager.java:444)

 at sandbox.highavailability.InvalidActiveNode$1.run(InvalidActiveNode.java:16)
 at com.kabira.platform.Transaction.execute(Transaction.java:484)
 at com.kabira.platform.Transaction.execute(Transaction.java:542)
 at sandbox.highavailability.InvalidActiveNode.main(InvalidActiveNode.java:18)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
 at java.lang.reflect.Method.invoke(Unknown Source)
 at com.kabira.platform.MainWrapper.invokeMain(MainWrapper.java:65)
 at
com.kabira.platform.highavailability.PartitionManager.definePartition(PartitionManager.java:810)
```

The amount of time to wait for a node to be come active before the `com.kabira.platform.ResourceUnavailableException` is thrown is controlled by the `nodeActiveTimeoutSeconds` configuration value. See the **TIBCO ActiveSpaces® Transactions Administration Guide** for details.

The `PartitionManager.enablePartitions(...)` and `partition.enable(...)` methods are synchronous - they block until any required object migrations are complete.

When enabling multiple partitions using the `PartitionManager.enablePartitions(...)` method, the order in which the partitions are enabled is undefined. The only guarantee is that all partitions have been enabled when `PartitionManager.enablePartitions(...)` returns. If ordering is required when enabling partitions, use the `partition.enable(...)` method.

The `PartitionManager.EnableAction` parameter to the enable methods controls how partitions are activated into the cluster.

- `JOIN_CLUSTER` - No object deletion or merging is done as part of activating partitions. This action is appropriate for initializing new nodes, or following hard failures that required shared memory to be removed.

- `JOIN_CLUSTER_PURGE` - Deletes any preexisting partitioned objects on the local node for the partition(s) being enabled before doing any object migration. This option is typically used after a node was gracefully removed from service and it is now being brought back online.

- `JOIN_CLUSTER_RESTORE` - Objects are both deleted and merged as part of activating the partitions. This option is used to restore partitions that were in a multi-master scenario. Objects are restored from the node specified in the restore from node partition property.

When restoring from a multi-master scenario, `ObjectNotUnique` exceptions can occur based on the `PartitionManager.EnableAction` value:

- `JOIN_CLUSTER` - An `ObjectNotUnique` exception can occur because shared memory is not cleared when the partition is enabled. If the restore from node has an object with the same key value, an `ObjectNotUnique` exception is thrown and object migration will terminate.

- `JOIN_CLUSTER_PURGE` - An `ObjectNotUnique` exception cannot occur because local shared memory is cleared before object migration occurs.

- `JOIN_CLUSTER_RESTORE` - An `ObjectNotUnique` exception cannot occur. The local object instance is deleted and copied from the restore from node.

Objects that exist only on a single node during a multi-master scenario are treated differently depending on the `PartitionManager.EnableAction` value. This inconsistency can occur if objects were created or deleted during a multi-master scenario. Assuming that an object exists on the node being restored, but not on the restore from node, the following summarizes the behavior:

- `JOIN_CLUSTER` - The object instance on the node being restored is orphaned - it is no longer a partitioned object since the node being restored from has no knowledge of this object.

- `JOIN_CLUSTER_PURGE` - The object on the node being restored is deleted since shared memory is cleared before the objects are migrated from the restore from node.

- `JOIN_CLUSTER_RESTORE` - The object on the node being restored is kept during the merge with the restore from node. This object appears to have been resurrected from the perspective of the restore from node, if it was previously deleted on that node.

Partition definition and enabling should be done in a separate transaction from any object creations that will use the new partition to minimize the risk of deadlocks. Creating objects in a partition that is not active causes a `com.kabira.platform.ResourceUnavailableException`, for example:

```
[A] com.kabira.platform.ResourceUnavailableException: partition 'Odd' is not active
[A]   at com.kabira.platform.ManagedObject.createSMObject(Native Method)
[A]   at
com.kabira.snippets.highavailability.FlintStone.<init>(PartitionDefinition.java:17)
[A]   at
com.kabira.snippets.highavailability.PartitionDefinition$2.run(PartitionDefinition.java:101)
[A]   at com.kabira.platform.Transaction.execute(Transaction.java:303)
[A]   at
com.kabira.snippets.highavailability.PartitionDefinition.main(PartitionDefinition.java:93)
[A]   at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
[A]   at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
[A]   at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
[A]   at java.lang.reflect.Method.invoke(Method.java:597)
[A]   at com.kabira.platform.MainWrapper.invokeMain(MainWrapper.java:49)
```

Example 7.2. Defining and enabling a partition

```
//     $Revision: 1.1.2.2 $
package com.kabira.snippets.highavailability;

import com.kabira.platform.Transaction;
import com.kabira.platform.Transaction.Rollback;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.highavailability.PartitionManager;
import static
com.kabira.platform.highavailability.PartitionManager.EnableAction.JOIN_CLUSTER;
import com.kabira.platform.highavailability.PartitionMapper;
import com.kabira.platform.highavailability.ReplicaNode;
import static com.kabira.platform.highavailability.ReplicaNode.ReplicationType.SYNCHRONOUS;

/**
 * This snippet shows how to define and enable new partitions
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = "A"
 * </ul>
 */
public class PartitionDefinition
{
    @Managed
    private static class FlintStone
    {

        FlintStone(String name)
        {
            this.name = name;
        }
        final String name;
    }

    //
    //  Partition mapper that puts object in partitions by name
    //
    private static class AssignPartitions extends PartitionMapper
    {
        @Override
        public String getPartition(Object obj)
        {
            FlintStone flintStone = (FlintStone) obj;

            return flintStone.name;
        }
    }

    /**
     * Main entry point
     *
     * @param args Not used
     */
    public static void main(String[] args)
    {
        new Transaction("Partition Definition")
        {
            @Override
            protected void run() throws Rollback
            {

                //
                //  Install the partition mapper
                //
                PartitionManager.setMapper(FlintStone.class, new AssignPartitions());
```

```
                //
                //  Set up the replicas
                //
                ReplicaNode[] fredReplicas = new ReplicaNode[]
                {
                    new ReplicaNode("B", SYNCHRONOUS)
                };
                ReplicaNode[] barneyReplicas = new ReplicaNode[]
                {
                    new ReplicaNode("B", SYNCHRONOUS)
                };

                //
                //  Define two partitions
                //
                PartitionManager.definePartition("Fred", null, "A", fredReplicas);
                PartitionManager.definePartition("Barney", null, "C", barneyReplicas);

                //
                //  Enable the partitions
                //
                PartitionManager.enablePartitions(JOIN_CLUSTER);

            }
        }.execute();

        new Transaction("Create Objects")
        {
            @Override
            protected void run() throws Rollback
            {
                //
                //  Create Fred and Barney
                //
                FlintStone fred = new FlintStone("Fred");
                FlintStone barney = new FlintStone("Barney");

                //
                //  Display assigned partitions
                //
                System.out.println(fred.name + " is in "
                    + PartitionManager.getObjectPartition(fred).getName());
                System.out.println(barney.name + " is in "
                    + PartitionManager.getObjectPartition(barney).getName());

            }
        }.execute();
    }
}
```

When Example 7.2 on page 118 is run it outputs the information in Example 7.3 on page 119.

Example 7.3. Partitioning example output

```
[A] Fred is in Fred
[A] Barney is in Barney
```

Partition migration

There are two mechanisms to migrate partitions:

- Programmatically using an API

- Administration tools.

Programmatic migration of partitions is covered here. See the **TIBCO ActiveSpaces® Transactions Administration Guide** for details on migrating partitions using the administration tools.

Migrating a partition involves optionally changing the partition properties, active node, or replica nodes associated with the partition. Partition migration must be done on the current active node for the partition. Depending on the changes to the partition, the active node may change, and object data may be copied to other nodes in the cluster.

Partition properties specified when a partition was defined can be overridden. This only affects the properties for the duration of the `partition.migrate()` execution. It does not change the default properties associated with a partition.

Calling `partition.migrate()` with any partitioned objects locked in the transaction causes the objects locked per transactions property to be treated as if it had a value of zero. This causes all objects in the partition to be locked in the calling transaction during the partition migration. It is strongly recommended that `partition.migrate()` always be called from a transaction with no partitioned objects locked.

Example 7.4 on page 120 shows an example of migrating a partition.

Example 7.4. Migrating a partition

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.highavailability;

import com.kabira.platform.Transaction;
import com.kabira.platform.Transaction.Rollback;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.highavailability.Partition;
import com.kabira.platform.highavailability.PartitionManager;
import static
com.kabira.platform.highavailability.PartitionManager.EnableAction.JOIN_CLUSTER;
import com.kabira.platform.highavailability.PartitionMapper;
import com.kabira.platform.highavailability.PartitionNotFound;
import com.kabira.platform.highavailability.ReplicaNode;
import static com.kabira.platform.highavailability.ReplicaNode.ReplicationType.SYNCHRONOUS;
import com.kabira.platform.property.Status;

/**
 * This snippet shows how to migrate a partition
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = "A"
 * </ul>
 */
public class PartitionMigration
{
    @Managed
    private static class AnObject
    {
    }

    //
    //  Partition mapper that maps objects to either Even or Odd
    //
    private static class UpdateMapper extends PartitionMapper
    {
        @Override
        public String getPartition(Object obj)
```

```
        {
            return PARTITION_NAME;
        }
    }

    /**
     * Main entry point
     *
     * @param args Not used
     * @throws java.lang.InterruptedException
     */
    public static void main(String[] args) throws InterruptedException
    {
        new Transaction("Initialization")
        {
            @Override
            protected void run() throws Rollback
            {
                //
                //  Install the partition mapper
                //
                PartitionManager.setMapper(AnObject.class, new UpdateMapper());

                //
                //  Define the partition
                //
                ReplicaNode[] replicas = new ReplicaNode[]
                {
                    new ReplicaNode("B", SYNCHRONOUS),
                    new ReplicaNode("C", SYNCHRONOUS)
                };
                PartitionManager.definePartition(PARTITION_NAME, null, "A", replicas);

                //
                //  Enable the partition
                //
                PartitionManager.enablePartitions(JOIN_CLUSTER);
            }
        }.execute();

        new Transaction("Create Object")
        {
            @Override
            protected void run() throws Rollback
            {
                m_object = new AnObject();

                //
                //  Get the partition for the object
                //
                Partition partition = PartitionManager.getObjectPartition(m_object);

                //
                //  Display the active node
                //
                System.out.println("Active node is " + partition.getActiveNode());
            }
        }.execute();

        new Transaction("Migrate Partition")
        {
            @Override
            protected void run() throws Rollback
            {
                //
                //  Get the partition for the object
                //
```

```
                Partition partition = PartitionManager.getObjectPartition(m_object);

                //
                //  Partition migration can only be done on the current
                //  active node
                //
                if (partition.getActiveNode().equals(
                    System.getProperty(Status.NODE_NAME)) == false)
                {
                    return;
                }

                //
                //  Mirgate the partition to make node C the active node.
                //
                //  Partition migration can only be from the current active node.
                //
                //  The default properties specified when the partition was
                //  defined are used.
                //
                ReplicaNode[] replicas = new ReplicaNode[]
                {
                    new ReplicaNode("B", SYNCHRONOUS),
                    new ReplicaNode("A", SYNCHRONOUS)
                };
                partition.migrate(null, "C", replicas);

                System.out.println("Migrated partition to node C");
            }
        }.execute();

        waitForMigration();

        new Transaction("Display Active Node")
        {
            @Override
            protected void run() throws Rollback
            {
                //
                //  Get the partition for the object
                //
                Partition partition = PartitionManager.getObjectPartition(m_object);

                //
                //  Display the active node again
                //
                System.out.println("Active node is " + partition.getActiveNode());
            }
        }.execute();
    }

    private static void waitForMigration() throws InterruptedException
    {
        System.out.println("Waiting for migration...");

        while (m_partitionFound == false)
        {
            Thread.sleep(5000);

            new Transaction("Wait for Migration")
            {
                @Override
                protected void run()
                {
                    try
                    {
                        Partition partition =
```

```
PartitionManager.getPartition(PARTITION_NAME);

                        //
                        //  Wait for partition to migrate to node C
                        //
                        m_partitionFound = partition.getActiveNode().equals("C");
                    }
                    catch (PartitionNotFound ex)
                    {
                        // not available yet
                    }
                }
            }.execute();
        }

        System.out.println("Partition migration complete.");
    }

    private static boolean m_partitionFound = false;
    private static AnObject m_object = null;
    private static final String PARTITION_NAME = "Partition Migration Snippet";
}
```

When Example 7.4 on page 120 is run it outputs the information in Example 7.5 on page 123 (annotation added and output reordered for clarity).

Example 7.5. Partition migration output

```
#
# Partition is active on node A
#
[C] Active node is A
[B] Active node is A
[A] Active node is A

#
# Nodes B and C are waiting for partition migration
#
[C] Waiting for migration...
[B] Waiting for migration...

#
# Node A migrates partition to node C
#
[A] Migrated partition to node C
[A] Waiting for migration...

#
# Migration completes
#
[C] Partition migration complete.
[B] Partition migration complete.
[A] Partition migration complete.

#
# Partition is now active on node C
#
[C] Active node is C
[B] Active node is C
[A] Active node is C
```

Initializing or restoring a node

The following steps should be taken during application initialization to initialize the high-availability services:

1. Install any required partition mappers. This must be done before any partitioned objects are created.

2. Wait for all nodes to be active that are required for the partition definitions on the local node.

3. Define and enable all partitions that should be known on the local node. This includes partitions for which the local node is the active or a replica node. It also includes any sparse partitions.

4. Enable application work in a separate transaction from the above steps.

Other than the installation of the partition mappers, all of these steps can be performed by the administrative tools, assuming that application work can be started through an administrative action, either an explicit command (start a channel), or load configuration. The Example 7.6 on page 124 snippet shows the pattern to be used for node initialization.

Restoring a node to service following a failure is identical to normal node initialization with the only possible exception being the `EnableAction` used when enabling partitions. During normal node initialization an `EnableAction` of `JOIN_CLUSTER` can be used because there are no objects in shared memory. However, in the case where a node is being restored, and shared memory contains objects that should be removed before re-joining the cluster, an `EnableAction` of `JOIN_CLUSTER_PURGE` should be used to remove the stale objects from shared memory.

Example 7.6. Node initialization

```
//     $Revision: 1.1.2.3 $
package com.kabira.snippets.highavailability;

import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.highavailability.Partition;
import com.kabira.platform.highavailability.PartitionManager;
import com.kabira.platform.highavailability.PartitionManager.EnableAction;
import com.kabira.platform.highavailability.PartitionMapper;
import com.kabira.platform.highavailability.PartitionMapper.Properties.Audit;
import com.kabira.platform.highavailability.ReplicaNode;
import com.kabira.platform.highavailability.ReplicaNode.ReplicationType;
import com.kabira.platform.property.Status;

/**
 * Initialize a node for high-availability
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainname</b> = Development</li>
 * </ul>
 * </p>
 */
public class LifeCycle
{
    @Managed
    private static class X
    {
        X(final String node)
        {
            m_node = node;
```

```
        }
        private final String m_node;
    };

    private static class Mapper extends PartitionMapper
    {
        @Override
        public String getPartition(Object o)
        {
            assert o instanceof X : o;
            return _PARTITION_NAME;
        }
    }

    public static void main(final String[] args)
    {
        installMapper();

        //
        //    Wait for all required nodes to come active
        //
        for (String n : _NODE_LIST)
        {
            waitForActive(n);
        }

        activatePartition();

        enableApplicationWork();
    }

    private static void installMapper()
    {
        new Transaction()
        {
            @Override
            protected void run()
            {
                //
                //    Verify that there are no unpartitioned instances
                //
                PartitionMapper.Properties properties
                    = new PartitionMapper.Properties(Audit.VERIFY_PARTIONING);
                PartitionManager.setMapper(X.class, new Mapper(), properties);
            }
        }.execute();
    }

    private static void waitForActive(final String node)
    {
        new Transaction()
        {
            @Override
            protected void run()
            {
                PartitionManager.waitForNode(node);
            }
        }.execute();

    }

    private static void activatePartition()
    {
        new Transaction()
        {

            @Override
```

```
            protected void run() throws Transaction.Rollback
            {
                ReplicaNode[] replicas = new ReplicaNode[2];
                replicas[0] = new ReplicaNode("B", ReplicationType.SYNCHRONOUS);
                replicas[1] = new ReplicaNode("C", ReplicationType.SYNCHRONOUS);

                PartitionManager.definePartition(
                    _PARTITION_NAME,
                    null,
                    "A",
                    replicas);
                Partition partition = PartitionManager.getPartition(_PARTITION_NAME);
                partition.enable(EnableAction.JOIN_CLUSTER);
            }
        }.execute();
    }

    private static void enableApplicationWork()
    {
        new Transaction()
        {
            @Override
            protected void run() throws Transaction.Rollback
            {
                new X(System.getProperty(Status.NODE_NAME));
            }
        }.execute();
    }

    private static final String _PARTITION_NAME = LifeCycle.class.getSimpleName();
    private static final String[] _NODE_LIST =
    {
        "A", "B", "C"
    };
}
```

Updating partition mapping

Objects can be dynamically reassigned to other partitions. This allows objects to be split or merged into partitions based on changing application states or available system resources.

There are two mechanisms to update the partition mapping:

- Programmatically using an API
- Administration tools

Programmatic re-partitioning is covered in this section. See the **TIBCO ActiveSpaces® Transactions Administration Guide** for details on re-partitioning objects using the administration tools.

Object re-partitioning requires either a *partition mapper* that makes dynamic partitioning decisions or the installation of a new partition mapper. A dynamic *partition mapper* can return different partition names for the same object when called at different times.

Re-partitioning is triggered by the `partition.update()` or `PartitionManager.repartitionInstance(...)` methods. These methods cause the *partition mappers* to be called for the objects being re-partitioned. Re-partitioning using the `partition.update()` method must be done on the active node for the partition.

The partition properties specified when a partition was defined can be overridden when using the `partition.update()` method. This only affects the properties for the duration of the `partition.update()` execution. It does not change the default properties associated with a partition.

Calling `partition.update()` with any partitioned objects locked in the transaction causes the objects locked per transactions property to be treated as if it had a value of zero. This causes all objects in the partition to be locked in the calling transaction during the re-partitioning. It is strongly recommended that `partition.update()` always be called from a transaction with no partitioned objects locked.

Example 7.4 on page 120 shows an example of splitting a partition and then migrating the new partition to a new node.

Example 7.7. Splitting a partition

```
//     $Revision: 1.1.2.5 $
package com.kabira.snippets.highavailability;

import com.kabira.platform.Transaction;
import com.kabira.platform.Transaction.Rollback;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.highavailability.Partition;
import com.kabira.platform.highavailability.PartitionManager;
import static
com.kabira.platform.highavailability.PartitionManager.EnableAction.JOIN_CLUSTER;
import com.kabira.platform.highavailability.PartitionMapper;
import com.kabira.platform.highavailability.ReplicaNode;
import static com.kabira.platform.highavailability.ReplicaNode.ReplicationType.SYNCHRONOUS;
import com.kabira.platform.property.Status;

/**
 * This snippet demonstrates how to split a partition by repartitioning and then
 * migrating one of the partitions
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainname</b> = "Development"
 * </ul>
 */
public class Repartition
{
    @Managed
    private static class A
    {
        A(int value)
        {
            m_value = value;
        }
        private final int m_value;
    }

    private static class DynamicMapper extends PartitionMapper
    {
        @Override
        public String getPartition(Object object)
        {

            if (m_split == false)
            {
                return _PARTITION_ORG;
            }

            assert object instanceof A : object;
            A a = (A) object;
```

```
            String name = null;
            if ((a.m_value % 2) == 0)
            {
                name = _PARTITION_ORG;
            }
            else
            {
                name = _PARTITION_NEW;
            }
            return name;
        }

        void setSplit()
        {
            m_split = true;
        }
        private boolean m_split = false;
    }

    /**
     * Main entry point
     *
     * @param args Not used
     * @throws java.lang.InterruptedException
     */
    public static void main(final String[] args) throws InterruptedException
    {

        initialize();

        //
        //    Create some objects on A
        //
        if (m_nodeName.equals("A") == true)
        {
            createObjects();
        }
        else
        {
            waitForPartitionActive(_PARTITION_ORG, "A");
        }

        displayPartitionCounts();

        //
        //    Add a new partition
        //
        addPartition();

        waitForPartitionActive(_PARTITION_NEW, "A");

        //
        //    Update the partition on node A
        //
        if (m_nodeName.equals("A") == true)
        {
            updatePartition();
        }

        //
        //    Migrate the new partition to node B
        //
        if (m_nodeName.equals("A") == true)
        {
            migratePartition();
        }
```

```
        else
        {
            waitForPartitionActive(_PARTITION_NEW, "B");
        }

        displayPartitionCounts();
    }

    private static void initialize()
    {
        new Transaction("Initialize")
        {
            @Override
            protected void run() throws Rollback
            {
                m_mapper = new DynamicMapper();
                PartitionManager.setMapper(A.class, m_mapper);

                ReplicaNode[] replicas = new ReplicaNode[]
                {
                    new ReplicaNode("B", SYNCHRONOUS),
                    new ReplicaNode("C", SYNCHRONOUS)
                };

                PartitionManager.definePartition(_PARTITION_ORG, null, "A", replicas);
                PartitionManager.enablePartitions(JOIN_CLUSTER);
            }
        }.execute();
    }

    private static void addPartition()
    {
        new Transaction("Add Partition")
        {
            @Override
            protected void run() throws Rollback
            {
                ReplicaNode[] replicas = new ReplicaNode[]
                {
                    new ReplicaNode("B", SYNCHRONOUS),
                    new ReplicaNode("C", SYNCHRONOUS)
                };

                PartitionManager.definePartition(_PARTITION_NEW, null, "A", replicas);
                Partition partition = PartitionManager.getPartition(_PARTITION_NEW);
                partition.enable(JOIN_CLUSTER);
            }
        }.execute();
    }

    private static void createObjects()
    {
        new Transaction("Create Objects")
        {
            @Override
            protected void run() throws Rollback
            {
                for (int i = 0; i < _NUMBER_OF_OBJECTS; i++)
                {
                    new A(i);
                }
            }
        }.execute();
    }

    private static void updatePartition()
    {
```

```
        new Transaction("Configure Partition Mapper")
        {

            @Override
            protected void run() throws Rollback
            {
                m_mapper.setSplit();
            }
        }.execute();

        new Transaction("Update Objects")
        {
            @Override
            protected void run() throws Rollback
            {
                //
                //    Update the original partition
                //
                Partition partition = PartitionManager.getPartition(_PARTITION_ORG);
                partition.update(null);
            }
        }.execute();
    }

    private static void migratePartition()
    {
        System.out.println("Migrating new partition to node B");

        new Transaction("Migrate Partition")
        {
            @Override
            protected void run() throws Transaction.Rollback
            {
                Partition partition = PartitionManager.getPartition(_PARTITION_NEW);

                //
                //    Migrate partition to node B
                //
                ReplicaNode[] replicas = new ReplicaNode[]
                {
                    new ReplicaNode("C", SYNCHRONOUS),
                    new ReplicaNode("A", SYNCHRONOUS)
                };
                partition.migrate(null, "B", replicas);
            }
        }.execute();
    }

    private static void displayPartitionCounts()
    {
        new Transaction("Display Partition Counts")
        {
            @Override
            protected void run()
            {
                for (Partition p : PartitionManager.getPartitions())
                {
                    System.out.println(p.cardinality()
                        + " objects in "
                        + p.getName()
                        + " active on node "
                        + p.getActiveNode());
                }
            }
        }.execute();
    }
```

```
    private static void waitForPartitionActive(
        final String name,
        final String nodename) throws InterruptedException
    {
        m_active = false;

        while (m_active == false)
        {
            new Transaction("Wait for Partition")
            {
                @Override
                protected void run()
                {
                    Partition partition = PartitionManager.getPartition(name);
                    assert partition != null : name;
                    if ((partition.getCurrentState() == Partition.State.ACTIVE)
                        && (partition.getActiveNode().equals(nodename)))
                    {
                        m_active = true;
                    }
                }
            }.execute();

            Thread.sleep(1000);
        }
    }

    private static final String m_nodeName = System.getProperty(Status.NODE_NAME);
    private static final int _NUMBER_OF_OBJECTS = 10;
    private static final String _PARTITION_ORG = Repartition.class.getSimpleName();
    private static final String _PARTITION_NEW = Repartition.class.getSimpleName() +
"_new";

    private static boolean m_active = false;
    private static DynamicMapper m_mapper;
}
```

When Example 7.7 on page 127 is run it outputs the information in Example 7.8 on page 131 (annotation added and output reordered for clarity).

Example 7.8. Partition split example output

```
#
#  Create 10 objects in original partition
#
[C] 0 objects in Repartition active on node A
[C] 0 objects in Repartition active on node A
[A] 10 objects in Repartition active on node A
[A] 0 objects in Repartition_new active on node A
[B] 0 objects in Repartition_new active on node A
[B] 10 objects in Repartition active on node A

#
#  Update and migrate the new partition to node B
#
[A] Migrating new partition to node B

#
#  Partition has been spilt and the new partition is now
#  active on node B
#
[A] 5 objects in Repartition_new active on node B
[A] 5 objects in Repartition active on node A
[C] 5 objects in Repartition_new active on node B
[C] 5 objects in Repartition active on node A
```

```
[B] 5 objects in Repartition_new active on node B
[B] 5 objects in Repartition active on node A
```

Disabled partition behavior

When a partition is disabled, any objects in the partition are migrated to the new active node for the partition. The Example 7.9 on page 132 demonstrates this behavior. An object is created in a partition, and then the active node for the partition is disabled. A method is executed on the object before and after the partition is disabled to show the method execution migrating from the initial active node to the new active node.

These steps are executed in the snippet:

1. A partition mapper is installed on all nodes.
2. A partition is defined and enabled on all nodes.
3. An object is created in the partition on node A.
4. All nodes in the cluster wait until the partition is defined and enabled on all nodes.
5. A method is executed on the object on all nodes in the cluster.
6. The partition is disabled on node A.
7. All nodes wait for the partition to no longer be active on node A.
8. A method is executed on the object on all nodes in the cluster.
9. The example waits to be stopped.

Example 7.9. Method execution in disabled partition

```
//     $Revision: 1.1.2.5 $
package com.kabira.snippets.highavailability;

import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
import com.kabira.platform.Transaction.Rollback;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.highavailability.Partition;
import com.kabira.platform.highavailability.PartitionManager;
import static
com.kabira.platform.highavailability.PartitionManager.DisableAction.LEAVE_CLUSTER;
import static
com.kabira.platform.highavailability.PartitionManager.EnableAction.JOIN_CLUSTER;
import com.kabira.platform.highavailability.PartitionMapper;
import com.kabira.platform.highavailability.PartitionNotFound;
import com.kabira.platform.highavailability.ReplicaNode;
import static com.kabira.platform.highavailability.ReplicaNode.ReplicationType.SYNCHRONOUS;
import com.kabira.platform.property.Status;

/**
 * This snippet demonstrates remote method invocation behavior for highly
 * available objects
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainname</b> = Development
 * </ul>
 */
```

```
public class MethodInvocation
{
    @Managed
    private static class X
    {
        //
        //  Returns node name where executed
        //
        String whereExecuted()
        {
            return System.getProperty(Status.NODE_NAME);
        }
    }

    //
    //  Partition mapper that puts objects in known partition
    //
    private static class NodeMapper extends PartitionMapper
    {

        @Override
        public String getPartition(Object obj)
        {
            return PARTITION_NAME;
        }
    }

    /**
     * Main entry point
     *
     * @param args Not used
     * @throws java.lang.InterruptedException
     */
    public static void main(String[] args) throws InterruptedException
    {
        //
        //  Initialize snippet
        //
        initialize();

        //
        //  Create an object on node A
        //
        if (m_nodeName.equals("A") == true)
        {
            new Transaction("Create object")
            {
                @Override
                protected void run() throws Rollback
                {
                    new X();
                }
            }.execute();
        }

        //
        //  Wait for partition to be active on node A
        //
        waitForPartition();

        //
        //  Execute object method
        //
        executeMethod();

        //
        //  Disable the partition on node A
```

```
        //
        disablePartition();

        //
        //  Wait for partition to no longer be active on node A
        //
        waitForDisable();

        //
        //  Execute object method
        //
        executeMethod();

        //
        //  Wait for termination
        //
        waitForStop();
    }

    private static void initialize()
    {
        new Transaction("Initialization")
        {
            @Override
            protected void run() throws Rollback
            {
                //
                //  Install the partition mapper
                //
                PartitionManager.setMapper(X.class, new NodeMapper());

                //
                //  Set up replica node list
                //
                ReplicaNode[] replicas = new ReplicaNode[]
                {
                    new ReplicaNode("B", SYNCHRONOUS),
                    new ReplicaNode("C", SYNCHRONOUS)
                };

                //
                //  Force replication
                //
                Partition.Properties properties =  new Partition.Properties();
                properties.forceReplication(true);

                //
                //  Define the partition
                //
                PartitionManager.definePartition(
                    PARTITION_NAME, properties, "A", replicas);

                //
                //  Enable the partition
                //
                Partition partition = PartitionManager.getPartition(PARTITION_NAME);
                partition.enable(JOIN_CLUSTER);
            }
        }.execute();
    }

    //
    //  Wait for termination
    //
    private static void waitForStop() throws InterruptedException
    {
        System.out.println("Waiting for stop...");
```

```
        while (true)
        {
            Thread.sleep(5000);
        }
    }

    //
    //  Disable partition
    //
    private static void disablePartition()
    {
        new Transaction("Disable partition")
        {
            @Override
            protected void run()
            {
                if (m_nodeName.equals("A") == false)
                {
                    return;
                }
                PartitionManager.disablePartitions(LEAVE_CLUSTER);
                System.out.println(PARTITION_NAME + " disabled.");
            }
        }.execute();
    }

    //
    //  Wait for partition to be defined
    //
    private static void waitForPartition() throws InterruptedException
    {
        System.out.println("Waiting for partition...");

        while (m_partitionFound == false)
        {
            Thread.sleep(5000);

            new Transaction("Wait for Partition")
            {
                @Override
                protected void run()
                {
                    try
                    {
                        Partition partition =
PartitionManager.getPartition(PARTITION_NAME);

                        //
                        //  Wait for partition to go active on node A
                        //
                        m_partitionFound = partition.getActiveNode().equals("A");
                    }
                    catch (PartitionNotFound ex)
                    {
                        // not available yet
                    }
                }
            }.execute();
        }

        System.out.println("Partition " + PARTITION_NAME + " found.");
    }

    //
    //  Wait for partition to move off of A
    //
```

```
    private static void waitForDisable() throws InterruptedException
    {
        System.out.println("Partition still active on node A...");

        while (m_activeOnA == true)
        {
            Thread.sleep(5000);

            new Transaction("Wait for Partition")
            {
                @Override
                protected void run()
                {
                    Partition p = PartitionManager.getPartition(PARTITION_NAME);

                    m_activeOnA = p.getActiveNode().equals("A");
                }
            }.execute();
        }

        System.out.println("Partition no longer active on node A");
    }

    //
    //  Execute method on objects
    //
    private static void executeMethod()
    {
        new Transaction("Execute Method")
        {
            @Override
            protected void run()
            {
                for (X x : ManagedObject.extent(X.class))
                {
                    System.out.println("Executed on: " + x.whereExecuted());
                }
            }
        }.execute();
    }

    private static Boolean m_activeOnA = true;
    private static Boolean m_partitionFound = false;
    private static final String m_nodeName = System.getProperty(Status.NODE_NAME);
   private static final String PARTITION_NAME = MethodInvocation.class.getSimpleName();
}
```

When the snippet is executed it outputs the information (annotation added and messages reordered for clarity) in Example 7.10 on page 136.

Example 7.10. Disabled partition output

```
#
#    All nodes are waiting for the partition to be enabled
#
[A] Waiting for partition...
[B] Waiting for partition...
[C] Waiting for partition...

#
#    The partition is found active by all of the nodes
#
[B] Partition Method Invocation found.
[A] Partition Method Invocation found.
[C] Partition Method Invocation found.
```

```
#
#   Method executed on object created in partition.
#   Method executes on node A (current active node).
#
[B] Executed on: A
[A] Executed on: A
[C] Executed on: A

#
#   Partition disabled on node A
#
[A] Method Invocation disabled.

#
#    All nodes waiting for partition to be disabled on node A
#
[B] Partition still active on node A...
[A] Partition still active on node A...
[C] Partition still active on node A...

#
#   Partition no longer active on node A
#
[B] Partition no longer active on node A
[A] Partition no longer active on node A
[C] Partition no longer active on node A

#
#   Method executed on object.  Now executes on node B (new active node)
#   Object no longer found on node A since partition migrated to
#   node B removing object from node A.
#
[B] Executed on: B
[C] Executed on: B

#
#   Snippet waiting for termination
#
[B] Waiting for stop...
[A] Waiting for stop...
[C] Waiting for stop...
```

Transparent failover

When the active node for a partition fails, it transparently fails over to the first replica for the partition. Any in-flight transactions not started on the node that failed transparently failover to the new active node for the partition. This ensures that any object modifications or method invocations complete successfully without any application action.

Example 7.11 on page 138 shows this behavior for a method invocation. These steps are executed in the snippet:

1. A partition mapper is installed on all nodes.
2. A partition is defined and enabled on all nodes with node `B` as the active node and node `C` as a replica.
3. An object is created in the partition on node `A`.
4. A method is executed on the object that is dispatched to node `B`.
5. The method causes node `B` to fail.

6. The partition falls over to the replica node C.

7. The method transparently executes on node C.

This snippet causes a hard node failure.

Example 7.11. Transparent failover

```
//     $Revision: 1.1.4.2 $
package com.kabira.snippets.highavailability;

import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.highavailability.PartitionManager;
import com.kabira.platform.highavailability.PartitionManager.EnableAction;
import com.kabira.platform.highavailability.PartitionMapper;
import com.kabira.platform.highavailability.ReplicaNode;
import com.kabira.platform.property.Status;

/**
 * Transparently fail-over following an active node failure.
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainname</b> = Development
 * </ul>
 */
public class TransparentFailover
{
    @Managed
    private static class A
    {
        void hello(final String fromNode)
        {
            String nodeName = System.getProperty(Status.NODE_NAME);
            System.err.println("INFO: On node " + nodeName
                + ":From node " + fromNode);

            //
            // Force the engine down if on active node
            //
            if (nodeName.equals(_ACTIVE_NODE) == true)
            {
                System.err.println("WARNING: Forcing down node " + nodeName);

                //
                // Just to let messages get to deployment tool
                //
                try
                {
                    Thread.sleep(1000);
                }
                catch (InterruptedException ex)
                {
                    // Don't care
                }
                Runtime.getRuntime().halt(0);
            }
        }
    }

    private static class Mapper extends PartitionMapper
    {
        @Override
```

```
        public String getPartition(Object o)
        {
            return _PARTITION_NAME;
        }
    }

    /**
     * Run the transparent failover snippet
     *
     * @param args None
     * @throws InterruptedException
     */
    public static void main(String[] args) throws InterruptedException
    {

        final String nodeName = System.getProperty(Status.NODE_NAME);

        initialize();

        if (nodeName.equals(_TEST_NODE) == false)
        {
            System.out.println("INFO: Sleeping on node " + nodeName);
            Thread.sleep(10000);
            return;
        }

        new Transaction()
        {
            @Override
            protected void run() throws Transaction.Rollback
            {
                A a = new A();
                a.hello(nodeName);
            }
        }.execute();
    }

    private static void initialize()
    {
        new Transaction()
        {
            @Override
            protected void run() throws Transaction.Rollback
            {
                PartitionManager.setMapper(A.class, new Mapper());
                ReplicaNode[] replicaNodes =
                {
                  new ReplicaNode(_REPLICA_NODE, ReplicaNode.ReplicationType.SYNCHRONOUS)

                };
                PartitionManager.definePartition(
                    _PARTITION_NAME, null, _ACTIVE_NODE, replicaNodes);
                PartitionManager.enablePartitions(EnableAction.JOIN_CLUSTER_PURGE);
            }
        }.execute();
    }

    private static final String _TEST_NODE = "A";
    private static final String _ACTIVE_NODE = "B";
    private static final String _REPLICA_NODE = "C";
    private static final String _PARTITION_NAME = "Transparent Failover";
}
```

When this snippet is executed it outputs the following messages (annotation added):

```
[C] INFO: Sleeping on node C
[B] INFO: Sleeping on node B
```

```
#
#  Method called from node A executes on node B
#
[B] INFO: On node B:From node A

#
#  Node B is forced down with an in-flight transaction
#
[B] WARNING: Forcing down node B

#
#  Method is transparently reexecuted on node C
#
[C] INFO: On node C:From node A
```

Partition state change notifiers

In certain cases an application may need to be notified of partition state changes. This is supported using the `com.kabira.platform.highavailability.PartitionNotifier` abstract class.

An application must create and register a partition notifier on each node where it is interested in receiving notifications of partition state changes. The `stateChange` method will be called for each state change for all partitions to which it is registered.

Example 7.12 on page 140 shows a simple example that monitors partition state changes. This example defines a single partition and then prints out all partition state changes. The partition state changes are triggered using administration tools.

This example does not exit until explicitly shutdown by terminating the deployment tool or using administration tools to stop the JVM.

Example 7.12. Monitoring partition state changes

```
//     $Revision: 1.1.2.4 $
package com.kabira.snippets.highavailability;

import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
import com.kabira.platform.highavailability.Partition;
import com.kabira.platform.highavailability.Partition.State;
import com.kabira.platform.highavailability.PartitionManager;
import com.kabira.platform.highavailability.PartitionNotifier;
import com.kabira.platform.highavailability.ReplicaNode;
import static com.kabira.platform.highavailability.ReplicaNode.ReplicationType.SYNCHRONOUS;

/**
 * Snippet on handling partition state changes.
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class StateChange
{
    /**
     * Partition state change notifier
     */
    private static class Notifier extends PartitionNotifier
    {
        @Override
```

```
        protected void stateChange(String partitionName, State oldState, State newState)

        {
            Partition partition = PartitionManager.getPartition(partitionName);

            String message = "Partition: "
                + partition.getName()
                + " transitioning "
                + "from "
                + oldState
                + " to "
                + newState
                + " now hosted on "
                + partition.getActiveNode();
            System.out.println(message);
        }
    }

    /**
     * Main entry point
     *
     * @param args Not used
     * @throws java.lang.InterruptedException
     */
    public static void main(String[] args) throws InterruptedException
    {
        final Notifier notifier = initialize();

        //
        //    Wait here for operator to shutdown example
        //
        waitForOperator();

        //
        //  Clean up partition notifier
        //
        new Transaction("Remove Partition Notifier")
        {
            @Override
            protected void run() throws Rollback
            {
                ManagedObject.delete(notifier);
            }
        }.execute();
    }

    //
    //  Wait for operator to shutdown example
    //
    private static void waitForOperator()
        throws InterruptedException
    {
        //
        // Wait until operator requests JVM to exit
        //
        while (true)
        {
            System.out.println("Waiting for operator...");
            Thread.sleep(5000);
        }
    }

    private static Notifier initialize()
    {
        return new Transaction("Create Partition Notifier")
        {
            @Override
```

```
            protected void run() throws Transaction.Rollback
            {
                m_notifier = new Notifier();

                ReplicaNode[] replicas = new ReplicaNode[]
                {
                    new ReplicaNode("B", SYNCHRONOUS),
                    new ReplicaNode("C", SYNCHRONOUS)
                };
                PartitionManager.definePartition(PARTITION_NAME, null, "A", replicas);

                //
                //  Partition is not enabled - use administration
                //  tools to enable partition
                //
                System.out.println("Enable the " + PARTITION_NAME + " partition.");

                //
                //  Register the partition notifier
                //
                Partition partition = PartitionManager.getPartition(PARTITION_NAME);
                partition.setNotifier(m_notifier);
            }

            Notifier initialize()
            {
                execute();
                return m_notifier;
            }

            private Notifier m_notifier = null;
        }.initialize();
    }

    private static final String PARTITION_NAME = "Notifier Snippet";
}
```

When Example 7.12 on page 140 is run it outputs the (annotation added) information in Example 7.13 on page 142.

Example 7.13. Partition state change output

```
#
# Partitions enabled on node A.  Partition comes active on A node
#
# administrator servicename=A enable partition name="Notifier Snippet"
#
Partition: Notifier Snippet transitioning from INITIAL to MIGRATING now hosted on A
Partition: Notifier Snippet transitioning from MIGRATING to ACTIVE now hosted on A

#
# Partitions disabled on node A.  Partition fails over to node B
#
# administrator servicename=A disable partition name="Notifier Snippet"
#
Partition: Notifier Snippet transitioning from ACTIVE to MIGRATING now hosted on A
Partition: Notifier Snippet transitioning from MIGRATING to ACTIVE now hosted on B
Partition: Notifier Snippet transitioning from ACTIVE to UNAVAILABLE now hosted on B

#
# Partition restored to node A from node B.
#
# administrator servicename=A define partition name="Notifier Snippet" activenode=A
replicas=B,C
# administrator servicename=A display partition name="Notifier Snippet"
#
```

```
Partition: Notifier Snippet transitioning from ACTIVE to MIGRATING now hosted on B
Partition: Notifier Snippet transitioning from MIGRATING to ACTIVE now hosted on A
```

Node state change notifiers

Node state change notifiers can be installed by applications to allow them to be notified when remote nodes change their state. Node state change notifiers can be used to manage local node resources, or to dynamically maintain partition definitions as nodes are added and removed from the cluster. Node state change notifiers are defined using the `com.kabira.platform.highavailability.NodeNotifier` abstract class.

An application must create and register a node notifier on each node where it is interested in receiving notifications of remote node state changes. The `active` method will be called every time a remote node comes active. The `active` method is also called for all active remote nodes that have already been discovered when the notifier is installed. The `unavailable` method will be called every time a remote node becomes unavailable.

Example 7.14 on page 143 shows a snippet that installs node notifiers on every node on which it is run. A partition is defined using the name of the local node, and the replica list for the partition is dynamically maintained as nodes come active. Partitioned objects are created in these partitions and are available to all other nodes in the cluster. This allows the other nodes to easily find and execute a remote method on all active nodes in the cluster.

This snippet does not exit until explicitly shutdown by terminating the deployment tool or using administrative tools to stop the JVM.

Example 7.14. Dynamically maintaining cluster-wide partitions

```
//     $Revision: 1.1.2.6.4.1 $
package com.kabira.snippets.highavailability;

import com.kabira.platform.KeyFieldValueList;
import com.kabira.platform.KeyManager;
import com.kabira.platform.KeyQuery;
import com.kabira.platform.LockMode;
import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Key;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.highavailability.NodeNotifier;
import com.kabira.platform.highavailability.Partition;
import com.kabira.platform.highavailability.PartitionManager;
import static
com.kabira.platform.highavailability.PartitionManager.EnableAction.JOIN_CLUSTER;
import com.kabira.platform.highavailability.PartitionMapper;
import com.kabira.platform.highavailability.ReplicaNode;
import com.kabira.platform.property.Status;

/**
 * Snippet showing how to handle node state changes.
 *
 * Each node defines a partition using its node name and adds and removes remote
 * nodes from the partition as they become active/inactive.
 *
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainname</b> = Development
 * </ul>
 */
```

```
public class NodeStateChange
{
 @Managed
 @Key(name = "ByName", fields =
 {
  "nodeName"
 }, unique = true, ordered = true)
 private static class NodeObject
 {
  final String nodeName;

  NodeObject(String nodeName)
  {
   this.nodeName = nodeName;
  }

  void remoteExecute(String caller)
  {
   System.out.println("remoteExecute executed on node "
    + m_nodeName
    + ", caller node was " + caller + ".");
  }

  @Override
  public String toString()
  {
   return "NodeObject: " + nodeName;
  }
 };

 private static class NodeObjectMapper extends PartitionMapper
 {
  @Override
  public String getPartition(Object obj)
  {
   NodeObject no = (NodeObject) obj;
   return no.nodeName;
  }
 }

 /**
  * Node state change notifier
  */
 private static class Notifier extends NodeNotifier
 {
  @Override
  protected void active(String remoteNode)
  {
   //
   //  Check to see if we already have seen this node.
   //
   Partition partition = PartitionManager.getPartition(m_nodeName);
   String[] currentNodeList = partition.getNodeList();

   for (String currentNode : currentNodeList)
   {
    if (currentNode.equals(remoteNode) == true)
    {
     return;
    }
   }

   //
   //  Add remote node to partition as a replica
   //
   String[] currentReplicas = partition.getReplicaNodes();
   ReplicaNode[] replicas = new ReplicaNode[currentReplicas.length + 1];
```

```
  for (int i = 0; i < currentReplicas.length; i++)
  {
   replicas[i] = partition.getReplicaNode(i);
  }
  replicas[currentReplicas.length]
   = new ReplicaNode(remoteNode, ReplicaNode.ReplicationType.SYNCHRONOUS);

  partition.migrate(null, partition.getActiveNode(), replicas);

           //
  //  Create object that will be pushed to all remote nodes.
  //
  createNodeObject(m_nodeName);

  System.out.println(m_nodeName + ": " + remoteNode + " active");
 }

 @Override
 protected void unavailable(String remoteNode)
 {
  System.out.println(m_nodeName + ": " + remoteNode + " unavailable");

  //
  //  Delete the remote instance, since we no longer can use it.
  //
  deleteNodeObject(remoteNode);

  //
  // The remote node will automatically get removed from the
  // partition definition as part of failover processing. So we
  // don't need to do anything here for our partition.
  //
 }

 private void createNodeObject(String node)
 {
  KeyManager<NodeObject> km = new KeyManager<>();
  KeyQuery<NodeObject> kq = km.createKeyQuery(NodeObject.class, "ByName");
  KeyFieldValueList kfvl = new KeyFieldValueList();
  kfvl.add("nodeName", node);
  kq.defineQuery(kfvl);
  kq.getOrCreateSingleResult(LockMode.WRITELOCK, null);
 }

 private void deleteNodeObject(String node)
 {
  KeyManager<NodeObject> km = new KeyManager<>();
  KeyQuery<NodeObject> kq = km.createKeyQuery(NodeObject.class, "ByName");
  KeyFieldValueList kfvl = new KeyFieldValueList();
  kfvl.add("nodeName", node);
  kq.defineQuery(kfvl);
  NodeObject no = kq.getSingleResult(LockMode.WRITELOCK);
  if (!ManagedObject.isEmpty(no))
  {
   ManagedObject.delete(no);
  }
 }
}

/**
 * Main entry point
 *
 * @param args Not used
 * @throws java.lang.InterruptedException
 */
public static void main(String[] args) throws InterruptedException
```

```
{
 final Notifier notifier = initialize();

 //
 //    Wait here for operator to shutdown example
 //
 waitForOperator();

 //
 //  Clean up node notifier
 //
 new Transaction("Remove Node Notifier")
 {
  @Override
  protected void run() throws Rollback
  {
   ManagedObject.delete(notifier);
  }
 }.execute();
}

private static Notifier initialize()
{
 return new Transaction("Initialize")
 {
  @Override
  protected void run() throws Transaction.Rollback
  {
   PartitionManager.setMapper(NodeObject.class, new NodeObjectMapper());
   PartitionManager.definePartition(m_nodeName, null, m_nodeName, null);
   PartitionManager.enablePartitions(JOIN_CLUSTER);

   m_notifier = new Notifier();
  }

  Notifier initialize()
  {
   execute();
   return m_notifier;
  }

  private Notifier m_notifier = null;
 }.initialize();
}

//
//  Wait for operator to shutdown example
//
private static void waitForOperator() throws InterruptedException
{
 while (true)
 {
  Thread.sleep(5000);

  new Transaction("Wait for Stop")
  {
   @Override
   protected void run()
   {
    callRemoteNodes();
   }
  }.execute();
  System.out.println(m_nodeName + ": Waiting for stop");
 }
}

private static void callRemoteNodes()
```

```
 {
  for (NodeObject no : ManagedObject.extent(NodeObject.class))
  {
   if (no.nodeName.equals(m_nodeName) == false)
   {
    no.remoteExecute(m_nodeName);
   }
  }
 }

 private static final String m_nodeName = System.getProperty(Status.NODE_NAME);
}
```

When Example 7.14 on page 143 is run it outputs the (annotation added) information in Example 7.15 on page 147.

Example 7.15. Dynamic partitions output

```
#
#    B & C are active on node A
#
[A] A: B active
[A] A: C active

#
#    A & B are active on node C
#
[C] C: A active
[C] C: B active

#
#    A & C are active on node B
#
[B] B: A active
[B] B: C active

#
#     Node A executed remote methods on C & B
#
[C] remoteExecute executed on node C, caller node was A.
[B] remoteExecute executed on node B, caller node was A.

#
#     Node C executed remote methods on A & B
#
[B] remoteExecute executed on node B, caller node was C.
[A] remoteExecute executed on node A, caller node was C.

#
#     Node B executed remote methods on A & C
#
[C] remoteExecute executed on node C, caller node was B.
[A] remoteExecute executed on node A, caller node was B.

     ...
```

Timers

TIBCO ActiveSpaces® Transactions provides support for highly available timers. Highly available timers are transparently migrated to new nodes during failover, restoration, and partition migration. Highly available timers are created by installing a *partition mapper* for the application defined

`com.kabira.platform.swtimer.TimerNotifier` types and mapping the timers into a valid partition with replica nodes defined.

Using asynchronous replication of timer operations may cause a loss of a timer notifications in the case of an active node failure.

Highly available timers are transactional. If a timer is executing on the currently active node, but it does not commit before a node failure, the timer will be executed on the node that assumes the work for the failed node.

Timer identifiers are a unique identifier for the timer on all nodes associated with the partition. The application can rely on this identifier being the same on all nodes. Timers are started using the number of seconds from the current time, i.e. a relative, not an absolute time. The timer fires when this time expires. The relative time is transmitted to the replica nodes for the timer and the current time on the replica nodes is used to calculate when the timer should fire. This minimizes the impact of clock drift between the active and replica nodes.

It is strongly recommend that a network time synchronization protocol be used to keep all system clocks consistent.

An application defined object is specified when creating a timer. This object is passed to the `timerNotifier` method when it is called. This provides a mechanism for an application to provide context for each timer that is started. It is strongly recommended that the application context object used to start the timer be in the same partition as the timer notifier. This ensures that the context object is still valid if failover or migration occur. However, it is legal to use context objects in different partitions, or even one that are not highly available.

Failover and migration

When an active node fails, any pending timers are automatically restarted on the new active node.

One-shot timers will only be executed on the failover node if they have not executed on the original active node before the failure. If the expiration timer expired during the migration to the failover node it will fire immediately, otherwise it will be executed at the initially scheduled time. The time is adjusted based on when the timer was started on the original active node.

A recurring timer will execute on the failover node at the next scheduled time. The initial execution on the failover node is adjusted based on when the timer last fired on the original active node. It will then continue to execute on the new active node using the initial interval. If a recurring timer was missed due to a delay between the initial active node failure and the failover node taking on the work, these scheduled timer executions will be dropped - there are no "makeup" executions for recurring timers.

Migrating a partition that contains active timers will cause the timer to be canceled on the old active node and restarted on the new active node. The same notifier execution rules as described for failover apply.

Timer example

Example 7.16 on page 149 shows how a highly available timer is created, migrated to a new node, and terminated.

Example 7.16. Highly available timers

```
//     $Revision: 1.1.2.5 $
package com.kabira.snippets.highavailability;

import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.highavailability.Partition;
import com.kabira.platform.highavailability.PartitionManager;
import static
com.kabira.platform.highavailability.PartitionManager.EnableAction.JOIN_CLUSTER;
import com.kabira.platform.highavailability.PartitionMapper;
import com.kabira.platform.highavailability.ReplicaNode;
import static com.kabira.platform.highavailability.ReplicaNode.ReplicationType.SYNCHRONOUS;
import com.kabira.platform.property.Status;
import com.kabira.platform.swtimer.TimerNotifier;

/**
 * This snippet shows how to use highly available timers.
 *
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainname</b> = Development
 * </ul>
 */
public class Timer
{
 /**
  * Partition mapper that maps objects to a specific partition
  */
 private static class TimerPartitionMapper extends PartitionMapper
 {
  @Override
  public String getPartition(Object obj)
  {
   return Timer.PARTITION_NAME;
  }
 }

 /**
  * Timer notifier
  * <p>
  * Timer notifier must be in same partition as the object passed to the
  * notifier.
  */
 private static class Notifier extends TimerNotifier
 {
  /**
   * Timer notifier
   *
   * @param timerId Timer identifier
   * @param object Timer context object
   */
  @Override
  public void timerNotify(String timerId, Object object)
  {
   Context c1 = (Context) object;
   c1.count += 1;

   System.out.println("Timer Id:" + timerId + " Value: " + c1.count);
  }
 }

 /**
  * Context passed to timer notifier
```

```
 */
@Managed
private static class Context
{
 int count;
}

/**
 * Main entry point
 *
 * @param args Not used
 * @throws java.lang.InterruptedException
 */
public static void main(String[] args) throws InterruptedException
{
 initialize();

 //
 // Start timer on node A
 //
 if (m_nodeName.equals("A") == true)
 {
  startTimer();
 }

       //
 //    Wait for timer to fire a few times
 //
 Thread.sleep(10000);

 //
 // Migrate the partition to node B
 //
 if (m_nodeName.equals("A") == true)
 {
  migratePartition();
 }

 //
 //    Wait for timer to fire a few times
 //
 Thread.sleep(10000);

       //
 //  Stop the timer on node B
 //
 if (m_nodeName.equals("B") == true)
 {
  stopTimer();
 }

}

private static void initialize()
{
 new Transaction("Initialize")
 {
  @Override
  protected void run() throws Transaction.Rollback
  {
              //
   //  Install a partition mapper
   //
   TimerPartitionMapper mapper = new TimerPartitionMapper();
   PartitionManager.setMapper(Notifier.class, mapper);
   PartitionManager.setMapper(Context.class, mapper);
```

```
                //
    //  Define and enable the test partition
    //
    ReplicaNode[] replicas = new ReplicaNode[]
    {
     new ReplicaNode("B", SYNCHRONOUS),
     new ReplicaNode("C", SYNCHRONOUS)
    };
    PartitionManager.definePartition(PARTITION_NAME, null, "A", replicas);
    PartitionManager.enablePartitions(JOIN_CLUSTER);
   }
  }.execute();
}

private static void startTimer()
{
 new Transaction("Start Timer")
 {
  @Override
  protected void run() throws Transaction.Rollback
  {
   Notifier notifier = new Notifier();
   Context c1 = new Context();

   System.out.println("Starting one second recurring timer");
   notifier.startRecurring(1, c1);
  }
 }.execute();
}

private static void stopTimer()
{
 new Transaction("Stop Timer")
 {
  @Override
  protected void run() throws Transaction.Rollback
  {
                //
   //    Stop timer - just delete the notifier
   //
   for (Notifier notifier : ManagedObject.extent(Notifier.class))
   {
    System.out.println("Stopping one second recurring timer");
    ManagedObject.delete(notifier);
   }
  }
 }.execute();
}

private static void migratePartition()
{
 System.out.println("Migrating partition to node B");

 new Transaction("Migrate Partition")
 {
  @Override
  protected void run() throws Transaction.Rollback
  {
   Partition partition = PartitionManager.getPartition(PARTITION_NAME);

   assert partition != null : PARTITION_NAME;

   //
   // Migrate partition to node B
   //

   ReplicaNode[] replicas = new ReplicaNode[]
```

```
    {
     new ReplicaNode("C", SYNCHRONOUS),
     new ReplicaNode("A", SYNCHRONOUS)
    };
    partition.migrate(null, "B", replicas);
   }
  }.execute();
 }

 private static final String PARTITION_NAME = "Timer Snippet";
 private static final String m_nodeName = System.getProperty(Status.NODE_NAME);
}
```

When Example 7.16 on page 149 is run the output in Example 7.17 on page 152 is seen (annotations added and non-essential output deleted).

Example 7.17. Highly available timer output

```
#
#     Timer started on node A
#
[A] Starting one second recurring timer

#
#     Timer notifier called on node A
#
[A] Timer Id:20799919691:107:0 Value: 1
[A] Timer Id:20799919691:107:0 Value: 2
[A] Timer Id:20799919691:107:0 Value: 3
[A] Timer Id:20799919691:107:0 Value: 4
[A] Timer Id:20799919691:107:0 Value: 5
[A] Timer Id:20799919691:107:0 Value: 6
[A] Timer Id:20799919691:107:0 Value: 7
[A] Timer Id:20799919691:107:0 Value: 8
[A] Timer Id:20799919691:107:0 Value: 9

#
#     Timer migrated to node B
#
[A] Migrating partition to node B

#
#    Timer notifier now called on node B
#
[B] Timer Id:20799919691:107:0 Value: 10
[B] Timer Id:20799919691:107:0 Value: 11
[B] Timer Id:20799919691:107:0 Value: 12
[B] Timer Id:20799919691:107:0 Value: 13
[B] Timer Id:20799919691:107:0 Value: 14
[B] Timer Id:20799919691:107:0 Value: 15
[B] Timer Id:20799919691:107:0 Value: 16
[B] Timer Id:20799919691:107:0 Value: 17
[B] Timer Id:20799919691:107:0 Value: 18

#
#    Timer stopped on node B
#
[B] Stopping one second recurring timer
```

Simulating a multi-master scenario

A multi-master scenario can be simulated using the `administrator` command or an API. This provides a mechanism for testing an application running in, and restoring from, a multi-master scenario.

Multi-master scenario simulation should only be used in a test environment. It should not be used in a production cluster.

A multi-master simulation is triggered using the *broadcast updates* partition property. When this property is set to `false`, any changes in a partition definition are not broadcast to the other nodes in a cluster when the partition is enabled. This allows a partition to be active on multiple nodes in a cluster. The broadcast updates property can be set using the `Partition.Properties` API or the `define partition` administrative command.

The broadcast updates property must be explicitly reset to true to re-enable normal cluster operation.

A multi-master scenario can only be simulated when redefining a partition on a node that is not currently in a partition's node list as either the active or a replica node. Example 7.18 on page 153 demonstrates the `administrator` commands to create a multi-master scenario.

Example 7.18. Multi-master simulation

```
//
//  Define and enable partition X
//
administrator servicename=A define partition name=X activenode=A replicas=B
administrator servicename=A enable partition name=X

//
//  Partition X status on active node A
//
administrator servicename=A display partition name=X
Partition Name = X
Partition State = Active
Partition Status = LocalEnabled
Last State Change Time = 2014-05-28 08:53:34
Active Node = A
Replica Nodes = B:synchronous
Replicate To Existing = false
Object Batch Size = 1000
Number Of Threads = 1
Restore From Node =
Mapped Types =
Broadcast Definition Updates = true

//
//  Partition X status on replica node B
//
administrator servicename=B display partition name=X
Partition Name = X
Partition State = Active
Partition Status = RemoteEnabled
Last State Change Time = 2014-05-28 08:53:34
Active Node = A
Replica Nodes = B:synchronous
Replicate To Existing = false
Object Batch Size = 1000
Number Of Threads = 1
```

```
Restore From Node =
Mapped Types =
Broadcast Definition Updates = true

//
// Redefine partition X on node C with broadcast updates disabled
//
administrator servicename=C define partition name=X activenode=C replicas=B
broadcastupdates=false
administrator servicename=C enable partition name=X

//
// Partition X is now active on node C with a replica of B
//
administrator servicename=C display partition name=X
Partition Name = X
Partition State = Active
Partition Status = LocalEnabled
Last State Change Time = 2014-05-28 09:00:05
Active Node = C
Replica Nodes = B:synchronous
Replicate To Existing = false
Object Batch Size = 1000
Number Of Threads = 1
Restore From Node =
Mapped Types =
Broadcast Definition Updates = false

//
//  Partition X remains active on node A since
//  broadcast updates were disabled
//
administrator servicename=A display partition name=X
Partition Name = X
Partition State = Active
Partition Status = LocalEnabled
Last State Change Time = 2014-05-28 08:57:25
Active Node = A
Replica Nodes = B:synchronous
Replicate To Existing = false
Object Batch Size = 1000
Number Of Threads = 1
Restore From Node =
Mapped Types =
Broadcast Definition Updates = true

//
//  Node B is now a replica for node C
//
administrator servicename=B display partition name=X
Partition Name = X
Partition State = Active
Partition Status = LocalEnabled
Last State Change Time = 2014-05-28 09:00:05
Active Node = C
Replica Nodes = B:synchronous
Replicate To Existing = false
Object Batch Size = 1000
Number Of Threads = 1
Restore From Node =
Mapped Types =
Broadcast Definition Updates = true
```

Failure exposure

This section describes possible data loss using the high-availability services:

Updates to objects only lose uncommitted creates, updates, and deletes to synchronous replication nodes when an active node fails. No application committed work can be lost.

Updates to objects using asynchronous replication lose any queued creates, updates, and deletes when an active node fails. Committed application work can be lost.

Inbound communications buffers that have not been processed by TIBCO ActiveSpaces® Transactions are lost. Some network buffers are configurable. A smaller network buffer size implies lower exposure to data loss.

You can avoid even this small risk of data loss if the client of the TIBCO ActiveSpaces® Transactions application has a protocol acknowledgement and includes retry logic if no acknowledgement is received. If the client application resends the request nothing is lost. However, care should be taken to handle duplicates.

8

Cluster Upgrades

This chapter describes how to upgrade nodes in a cluster independently without requiring a cluster outage. The specific steps required by an application to support mismatched class versions are:

1. Modify the classes - see the section called “Changing a class” on page 159.
2. Implement the object mismatch trigger if required - see the section called “Object mismatch interface” on page 159.
3. Run the upgrade utility to generate an *upgrade plan* - see the section called “Upgrade utility” on page 163.
4. Deploy the upgraded application - see the section called “Putting it all together” on page 166.

Supported changes

This section summarizes the managed class changes that are supported - both transparent and non-transparent. Unsupported changes are summarized in the section called “Unsupported changes” on page 158. The upgrade utility discussed in the section called “Upgrade utility” on page 163 audits all of the changes summarized in these sections and provides details on any incompatible changes being made.

Non-managed classes can be changed as required. These changes have no impact on other nodes in the cluster.

Managed classes being changed must be partitioned classes in a partition that has at least one replica node. This is to ensure that there is no data loss during the upgrade process. An attempt to upgrade a non-partitioned class, or a partitioned class without a replica node, will fail during the upgrade process. See the **TIBCO ActiveSpaces® Transactions Administration** for details on the upgrade process.

Transparent changes

Transparent changes to application classes do not require any application support. TIBCO ActiveSpaces® Transactions supports these changes transparently:

- Adding a field - the default value of the field is zero or empty, depending on the field type, when populating the field on the node with the new version.
- Removing a field - the default value of the field is set to zero or empty, depending on the field type, before sending it to a node with a down-rev version.
- Removing one or more keys.
- Removing fields from an existing multi-part key definition.
- Changing a field modifier from a non-static to a static final field (equivalent to removing a field).
- Changing a field modifier from a static final to a non-static final field (equivalent to adding a field).
- Adding a new method.

Transparent changes can also be handled explicitly by the application (except for removing a key or a key field). This allows an application to override the default behavior. For example an application might chose to set a non-zero default value for an integer field.

Removing a key, or a field in a multi-part key cannot be handled by the application. If a distributed query detects that a key was removed, or it's fields changed, it is ignored. No query results will be returned from a node that does not match the key definition of the node that initiated the distributed query.

Non-transparent changes

Non-transparent changes require the application developer to explicitly code the behavior that to map from a previous version to the new version. This mapping is done in an `ObjectMismatchTrigger` which is described in the section called “Object mismatch interface” on page 159. These are the non-transparent changes:

- Changing a field type. This includes changing a field type from a primitive to its wrapper class. For example, changing a field from an `int` to an `Integer` type is a field type change.
- Changing inheritance hierarchies.

Unsupported changes

These class changes are not supported:

- Changing a field modifier from non-final to final.
- Adding a key.
- Adding new fields to an existing key definition.
- Changing the type of a key field.
- Removing a method.

- Changing the signature of a method.
- Adding or removing values from an enumeration.
- Removing an enumeration referenced in a field, even if the field was removed.
- Removing a class referenced in a field, even if the field was removed.
- Removing `@Managed` from an existing class.
- Adding `@Managed` to an existing class.

Changing a class

Managed classes that are being changed must:

- implement `java.io.Serializable`.
- define a `private static final long serialVersionUID`.
- optionally implement `com.kabira.platform.ObjectMismatchTrigger`.

Previous versions of the class being changed may, or may not, have met any of these requirements. This is legal and allows classes to be changed, without previously meeting these requirements.

The details on how the `serialVersionUID` is used for class versioning is found in the **TIBCO ActiveSpaces® Transactions Architect's Guide**.

The `com.kabira.platform.ObjectMismatchTrigger` interface is required only if there are non-transparent changes.

Object mismatch interface

The `ObjectMismatchTrigger` interface is show in Example 8.1 on page 159.

Example 8.1. ObjectMismatchTrigger interface

```
public interface ObjectMismatchTrigger
{
    public void writeObjectToStream(
        ManagedObjectStreamClass remoteClassDescriptor,
        ObjectOutputStream out) throws IOException;

    public void readObjectFromStream(
        ManagedObjectStreamClass remoteClassDescriptor,
        ObjectInputStream in) throws IOException;
}
```

It defines these methods:

- `writeObjectToStream` - used to transform an object from a new to an old version.
- `readObjectFromStream` - used to transform an object from an old to a new version.

These methods are only called on the node where the new class version is installed. In all cases, the `remoteClassDescriptor` parameter contains a description of the old version of the class.

For example, in Figure 8.1, the `remoteClassDescriptor` parameter always contains a description of `O1`.

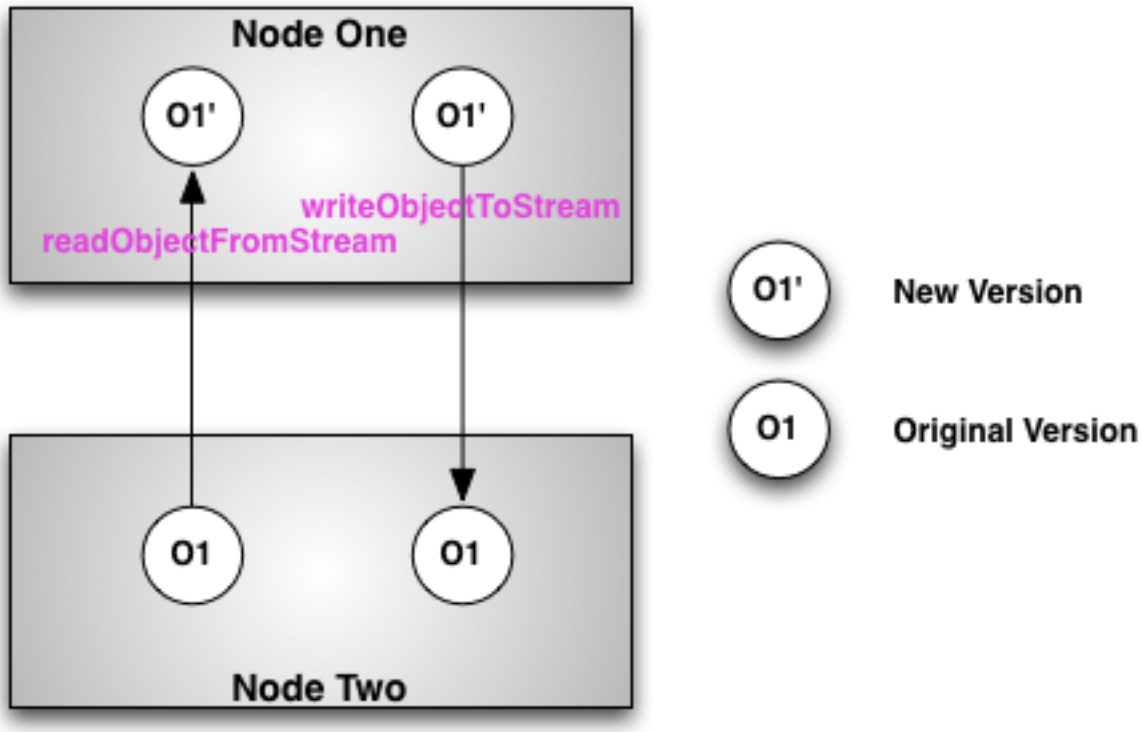

Figure 8.1. Object mismatch method invocation

Versions

The `ManagedObjectStreamClass` provides access to the remote `serialVersionUID` for a class using the `getSerialVersionUID()` method. The version information can be used to perform conditional mapping based on the actual class version on a remote node. This makes it possible to support multiple versions across a cluster. The `ObjectMismatchTrigger` interface can use the version information associated with the remote class to conditionally map the differences.

If no `serialVersionUID` is set in a class, `getSerialVersionUID()` returns a value of `0L`. To ensure that class version numbers are unambiguous, a `serialVersionUID` value of `0L` should never be used.

When supporting multiple deployed versions of a class, the upgrade utility must be run against the new class version and the most current old version. Not doing this may cause unresolved version mismatches are runtime.

Field mapping

The `ManagedObjectStreamClass` contains an ordered array of all fields, starting with the top most parent, to the current child class where an `ObjectMismatchTrigger` method is called. These fields must be processed in order. This is done using the `ManagedObjectStream-Class.getFields()` method.

The general approach to field mapping using the `ManagedObjectStreamClass` class is to iterate over all of the fields returned by the `ManagedObjectStreamClass` and map them into the new field definitions. The `ManagedObjectStreamClass` always contains the field definitions for the old class version.

An alternative approach to field mapping is to use the `ObjectInputStream.GetField` and `ObjectOutputStream.PutField` classes. These classes provide random access to fields by name. Again, the `ObjectInputStream.GetField` and `ObjectOutputStream.PutField` always contain the field definitions for the old class version.

The use of the `ManagedObjectStreamClass` to directly serialize and de-serialize a stream will be more efficient than using `ObjectInputStream.GetField` and `ObjectOutputStream.PutField` since fewer allocations and data copies are required.

See the section called “Inheritance” on page 161 for details on how inheritance affects field processing. A field mapping example is shown in Example 8.3 on page 164. This example shows the use of both the `ManagedObjectStreamClass` and the `ObjectInputStream.GetField` class.

Inheritance

Classes that extend other classes, must ensure that the parent class processes the object stream before the child. This can be done a couple of ways:

- Use `super` to call the parent's implementation of the `ObjectMismatchTrigger` method. This will only work if the inheritance hierarchy hasn't changed between the old and new class versions.
- Directly set the parent's field values. This is the only option if the inheritance hierarchy has changed between the old and new class versions. This does imply that a child must have access to all parent class fields either because they are `protected`, or there are setters available.

When an `ObjectMismatchTrigger` method is executed, the method always has complete access to any parent class fields in the old class version.

`ObjectMismatchTrigger` methods are only called on the leaf type of an inheritance hierarchy. They are not called on any of the parent types.

Here is an example using a call to `super` to populate a parent's fields:

```
//
//  Both old and new class versions extend Base
//
class Child extends Base
{
    public void readObjectFromStream(
        ManagedObjectStreamClass remoteClassDesc,
        ObjectInputStream in) throws IOException
    {
        //
        // Call the parent to process its fields (like a constructor,
        // this must be called before reading the child fields).
        //
        super.readObjectFromStream(remoteClassDesc, in);

        //
        // Process child fields
        //
    }
}
```

Here is another example, where the class hierarchy has changed between the old and new class version.

```
// VERSION 1:
@Managed
class InheritBase implements ObjectMismatchTrigger, Serializable
{
    int    int_val;
}
class InheritChild extends InheritBase
{
    byte []    byte_array;
}
class Inherit extends InheritChild
{
    String []   str_array;
}

//
//  VERSION 2: collapsed the class hierachy into a single class
//
@Managed
class Inherit implements ObjectMismatchTrigger, Serializable
{
    int        int_val;
    byte []    byte_array;
    String []  str_array;

    public void readObjectFromStream(
        ManagedObjectStreamClass remoteClassDesc,
        ObjectInputStream in) throws IOException
    {
        //
        // Process all fields directly here, this includes
        // the parent class fields (InheritBase & InheritChild)
        // in VERSION 1
        //
    }
}
```

Final fields

Reflection must be used to set `final` fields in a class in the `readObjectFromStream` method. The Java language prohibits the setting of `final` fields after an object is created - but that is exactly what is required when mapping a `final` field in the `readObjectFromStream` method.

To set a `final` field in `readObjectFromStream` requires code like the following, where field `id` is defined as `final`.

```
try
{
    java.lang.reflect.Field f = this.getClass().getDeclaredField("id");
    f.setAccessible(true);
    f.set(this, val);
}
catch (NoSuchFieldException ex)
{
    throw new IOException(ex);
}
catch (IllegalAccessException ex)
{
    throw new IOException(ex);
}
```

Mapping errors

Errors processing an object stream are reported using a `com.kabira.platform.DataError` exception. This exception contains information on the value that was in error. For example:

```
com.kabira.platform.DataError: Stream Corrupted:
   invalid type code: 0x77, expected type code: 0x70
```

The type code values are defined by the Java Serialization specification Object Serialization Stream Protocol
[http://download.oracle.com/javase/6/docs/platform/serialization/spec/protocol.html#10152] chapter. The constant definitions are also found in the java.io.ObjectStreamConstants [http://download.oracle.com/javase/6/docs/api/java/io/ObjectStreamConstants.html] javadoc.

The error above indicates that a `TC_BLOCKDATA` type code was seen when a `TC_NULL` was expected. This error was caused by attempting to write an integral type (`out.writeBytes()`) when a string was expected (`out.writeObject()`).

Upgrade utility

The upgrade utility is used to generate an *upgrade plan* based on a detailed analysis of class changes. The generated *upgrade plan* is used to deploy the upgraded class files. A complete reference for the upgrade utility can be found in the section called “Upgrade utility” on page 256. Example 8.2 on page 163 shows a simple `Person` class definition.

Example 8.2. Initial class definition

```
package com.kabira.snippets.upgrade;

import com.kabira.platform.annotation.Managed;

@Managed
public class Person
{
    Person()
    {
        name = "Patti Smith";
        age = 64;
    }
    String  name;
    short   age;
}
```

Example 8.3 on page 164 shows an updated `Person` class with these changes:

- Change the `name` field to two fields - `first` and `last`.
- Change the type of the `age` field to an `Integer`.
- Add a new base class - `Animal`.

The new `Person` class was also updated to implement `Serializable`, `ObjectMismatchTrigger`, and to add a `serialVersionUID` as described in the section called “Changing a class” on page 159.

Example 8.3. Updated class definition

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.upgrade;

import com.kabira.platform.ManagedObjectStreamClass;
import com.kabira.platform.ObjectMismatchTrigger;
import com.kabira.platform.annotation.Managed;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.ObjectStreamField;
import java.io.Serializable;


/*
//
//   Original class
//
@Managed
public class Person
{
    Person()
    {
        name = "Patti Smith";
        age = 64;
    }
    String  name;
    short   age;
}
 */

//
//  New parent type
//
@Managed
class Animal
{
    Animal()
    {
        type = "Human";
    }
    String type;
}

//
//   Updated class definition
//
public class Person extends Animal
    implements Serializable, ObjectMismatchTrigger
{
    Person()
    {
        first = "Patti";
        last = "Smith";
        age = 64;
    }

    @Override
    public void writeObjectToStream(
        ManagedObjectStreamClass remoteClassDescriptor,
        ObjectOutputStream out) throws IOException
    {
        System.out.println("writeObjectToStream: "
            + remoteClassDescriptor.getName());

        //
```

```
        //  Output old and new version numbers
        //
        System.out.println("\tOld Class Version: "
            + remoteClassDescriptor.getSerialVersionUID());
        System.out.println("\tNew Class Version: " + serialVersionUID);

        //
        //  Write fields to old version of class using ManagedObjectStreamClass
        //
        for (ObjectStreamField f : remoteClassDescriptor.getFields())
        {
            //
            //  If name field, concatenate first and last name in new version
            //
            if (f.getName().equals("name"))
            {
                System.out.println("\t\tname field: " + first + " " + last);
                out.writeObject(first + " " + last);
            }
            //
            //  If age field, map to short.  This may truncate the age of
            //  _very_ old people
            //
            else if (f.getName().equals("age"))
            {
                System.out.println("\t\tage field: " + age);
                short oldAge = age.shortValue();
                out.writeShort(oldAge);
            }
        }
    }

    @Override
    public void readObjectFromStream(
        ManagedObjectStreamClass remoteClassDescriptor,
        ObjectInputStream in) throws IOException
    {
        System.out.println("readObjectFromStream: "
            + remoteClassDescriptor.getName());

        //
        //  Output old and new version numbers
        //
        System.out.println("\tOld Class Version: "
            + remoteClassDescriptor.getSerialVersionUID());
        System.out.println("\tNew Class Version: " + serialVersionUID);

        //
        //  Read fields from the old version of the class using
        //  ObjectInputStreamGetField to access the fields
        //
        try
        {
            ObjectInputStream.GetField fields = in.readFields();

            //
            //  Get name field
            //
            String name = (String) fields.get("name", "");

            System.out.println("\t\tname: " + name);
            String[] values = name.split(" ");
            first = values[0];
            last = values[1];

            //
            //  Get age field
```

```
            //
            Short previousAge = fields.get("age", (short)0);
            age = previousAge.intValue();
            System.out.println("\t\tage: " + age);
        }
        catch (ClassNotFoundException ex)
        {
            System.out.println("ERROR: " + ex.getMessage());
        }

        //
        //  Initialize new parent field
        //
        this.type = "Human";
    }
    private static final long serialVersionUID = 1L;
    String first;
    String last;
    Integer age;
}
```

Running the upgrade utility on the old and new `Person` classes generates this output:

```
java -jar upgrade.jar current=original replacement=dist
1427 classes processed in 0.088 seconds.

Possible problems:
==================

com.kabira.snippets.upgrade.Animal
    Not found in the current JAR files. If this is a
    new class, this is not a problem. If this is not a
    new class, the class cannot be found in the current
    JAR files and must be provided.


Changed classes:
================

Name: com.kabira.snippets.upgrade.Person
Current serialVerisonUID: Not Set
Replacement serialVersionUID: 1
Mismatch handling: Application
Transparent changes:
    Removed field name.
    Added field last.
    Added field first.
Non-transparent changes:
    Inheritance modified.
    Type of field age changed.

Generated upgrade file: upgrade.111122
```

The possible problems section indicates that the `com.kabira.snippets.upgrade.Animal` class was not found in the previous version. This is expected because this is a new class. The upgrade utility reports this as a possible problem because it cannot differentiate between a new class, or one that was not provided in the current JAR files.

Putting it all together

This section completes this chapter by upgrading a cluster with the new class. The class being upgraded is the Person class discussed in the section called “Upgrade utility” on page 163. The simple

application show in Example 8.4 on page 167 is being deployed. This application performs these steps:

1. Define and enable a partition.

2. Create a `Person` object.

3. Wait for the operator to shut down the application.

Example 8.4. Upgrade application

```
//     $Revision: 1.1.2.2 $
package com.kabira.snippets.upgrade;

import com.kabira.platform.highavailability.ReplicaNode;
import static com.kabira.platform.highavailability.ReplicaNode.ReplicationType.*;
import com.kabira.platform.Transaction;
import com.kabira.platform.Transaction.Rollback;
import com.kabira.platform.highavailability.PartitionMapper;
import com.kabira.platform.highavailability.PartitionManager;
import static com.kabira.platform.highavailability.PartitionManager.EnableAction.*;

//
//  Partition mapper that puts all objects in a known partition
//
class Mapper extends PartitionMapper
{
    final static String PARTITION_NAME = "Upgrade";

    @Override
    public String getPartition(Object obj)
    {
        return PARTITION_NAME;
    }
}

/**
 * Define a partition and create an instance of an object that will be upgraded
 *
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = "A"
 * </ul>
 */
public class Upgrade
{
    /**
     * Main entry point
     *
     * @param args  Not used
     * @throws InterruptedException
     */
    public static void main(String[] args) throws InterruptedException
    {
        new Transaction("Upgrade")
        {
            @Override
            protected void run() throws Rollback
            {
                PartitionManager.setMapper(Person.class, new Mapper());
                ReplicaNode [] replicas = new ReplicaNode []
                {
                    new ReplicaNode("B", SYNCHRONOUS),
                    new ReplicaNode("C", SYNCHRONOUS)
```

```
                };
                PartitionManager.definePartition(
                    Mapper.PARTITION_NAME,
                    null,
                    "A",
                    replicas);
                PartitionManager.enablePartitions(JOIN_CLUSTER);
            }
        }.execute();

        new Transaction("Create Object")
        {
            @Override
            protected void run() throws Rollback
            {
                new Person();
            }
        }.execute();

        waitForStop();
    }

    //
    //  Wait for termination
    //
    private static void waitForStop() throws InterruptedException
    {
        System.out.println("Waiting for stop...");

        while (true)
        {
            Thread.sleep(5000);
        }
   }
}
```

The application is deployed by copying the JAR file containing the upgrade application to a deployment directory and using the deployment tool in detached mode to deploy the application. These steps are shown for a deployment directory located in `/Volumes/kabira/ast/deploy`.

```
#
#   Copy JAR file to deployment directory.
#   The snippet.jar file is used which is the output of building
#   the entire snippets project.
#
cp snippets.jar /Volumes/kabira/ast/deploy

#
#   Deploy the application to node A using the deployment tool.
#
#   NOTE: hostname will be specific to your environment.
#
java -jar deploy.jar detach=true username=guest password=guest \
     hostname=172.16.208.128 adminport=2000 domainnode=A \
     com.kabira.snippets.upgrade.Upgrade

INFO: deploy.jar version: [TIBCO TIBCO ActiveSpaces® Transactions
 2.2.0 (build 120302)]
      starting at [Sat Mar 10 09:10:41 PST 2012]
INFO: node [A] version: [TIBCO TIBCO ActiveSpaces® Transactions
 2.2.0 (build 120302)]
INFO: node [A] JVM remote debugger agent listening on port [21267] ...
INFO: Starting application [com.kabira.snippets.upgrade.Upgrade] ...
[A] WARNING: loopback ip address choosen, this agent may not connect to remote agents
[A] ip_address=127.0.0.1 port=50004
[A] Listening for transport dt_socket at address: 21267
```

```
[A] INFO: JMX Management Service started at:
[A]   kabira-server:2099
[A]   172.16.208.130:2099
[A]   service:jmx:rmi:///jndi/rmi://kabira-server:2099/jmxrmi
[A] Waiting for stop...
INFO: Component [com.kabira.snippets.upgrade.Upgrade] started in detached mode.
```

Displaying the initial version of a `Person` object in shared memory on node `A` shows the values in Figure 8.2.

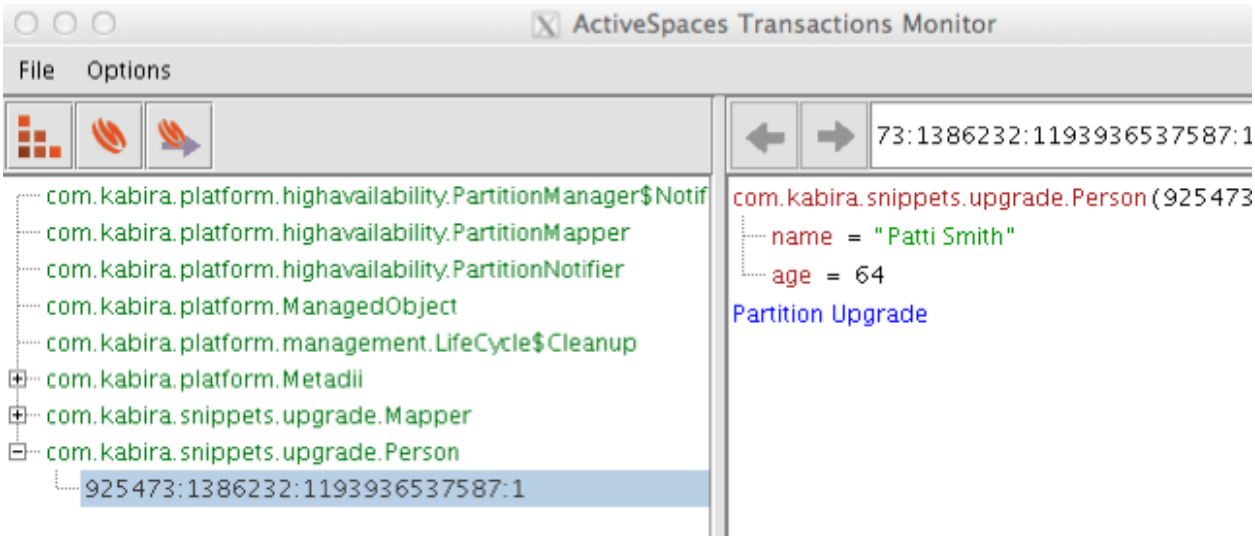

Figure 8.2. Initial version of person object

The replicated version of this object has the same values on replica nodes `B` and `C`.

The `Person` class is updated and a new version of the `snippets.jar` file is generated and deployed into the deployment directory after generating an *upgrade plan* as shown in the section called "Upgrade utility" on page 163. Node A is then upgraded using the TIBCO ActiveSpaces® Transactions Administrator `Upgrade or Restore...` node dialog shown in Figure 8.3.

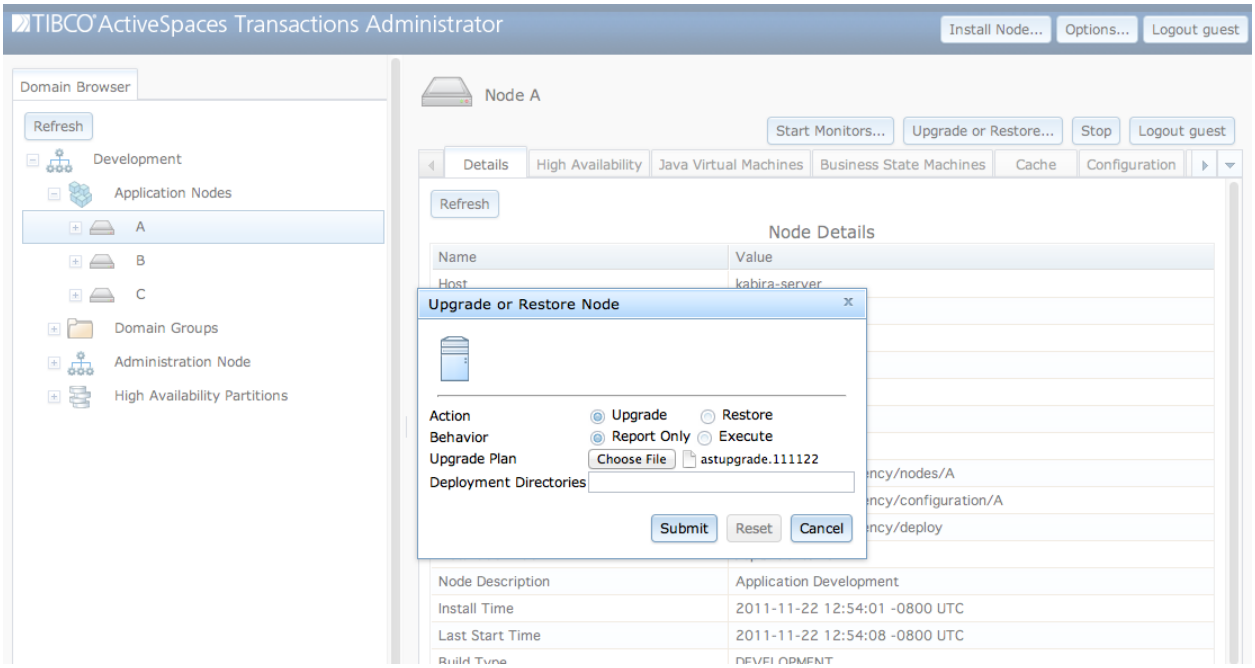

Figure 8.3. Upgrade node

When the upgrade completes an upgrade report is displayed as shown below.

```
[A] Starting upgrade ...
[A]
[A] Removing node A from the cluster ...
[A]
[A] Stopping node A ...
[A]
[A]   Stopping node
[A]     node::administration stopping ...
[A]  Ok
[A]     distribution::distribution stopping ... Ok
[A]     application::com_kabira_snippets_upgrade_Upgrade1 stopping ... Ok
[A]   Node stopped
[A]
[A] Starting type replacement on node A ...
[A]   running
[A]   running
[A]   running
[A]   complete
[A]
[A] Starting node A ...
[A]
[A]   Node A is already running.
[A]
[A]   Node A is configured to use DEVELOPMENT executables
[A]   Node A shared memory size is 512Mb
[A]   Node A path: /opt/kabira/run/ast/nodes/A
[A]   Node A host: kabira-server
[A]
[A]   System Coordinator Host: All Interfaces
[A]   System Coordinator Port: 2001
[A]
[A]   Web Server Host: All Interfaces
[A]   Web Server Port: 12796
[A]
[A]   Waiting for application to start ......
```

```
[A]
[A]   Components started
[A]   Loading configurations
[A]   Auditing security configuration
[A]   Host: localhost
[A]   Administration Port: 2001
[A]   Service Name: "A"
[A]   Node Name: "A"
[A]
[A] Updating partitions on node A ...
[A]   Partition Name = Upgrade
[A]   Active Node = B
[A]   Replica Nodes = C,A
[A]
[A] Enabling partitions on node A ...
[A] ...
[A] ...
[A] ...
[A] ...
[A] ...
[A]   Partition Name = Upgrade
[A]   Partition State = Active
[A]   Last State Change Time = 2011-10-07 13:29:08
[A]   Active Node = B
[A]   Replica Nodes = C:synchronous,A:synchronous
[A]
[A] Completed upgrade.
[A]
[A] Summary:
[A]   Node: A
[A]   Node state when command started: Running
[A]   Action: upgrade
[A]   Execute: true
[A]   Started: Fri Oct  7 13:28:40 PDT 2011
[A]   Finished: Fri Oct  7 13:29:09 PDT 2011
[A]   Classes File: /var/tmp/kmphpCpuNvx/upgrade.111007
[A]   Original Deployment Directories: /opt/kabira/run/ast/deploy
[A]   New Deployment Directories: /opt/kabira/run/ast/deploy
[A]
[A] Classes upgraded:
[A]   Class name: com.kabira.snippets.upgrade.Person
[A]   Current version: Not Set
[A]   New version: 1
[A]
[A] Active partitions migrated during upgrade:
[A]   Partition Name: Upgrade
[A]   Number of Objects: 1
[A]   New Active Node: B
```

At this point there are two different versions of the `Person` class deployed in the cluster. This can be seen in Figure 8.4. Notice that node `A` is reporting a local version of 1 and no remote version, while node `B` and `C` are reporting no local version, but a remote version of 1 for node `A`. This is because the new class deployed on node `A` defined a `serialVersionUID` field with a value of 1, while the old class version did not define a `serialVersionUID` field.

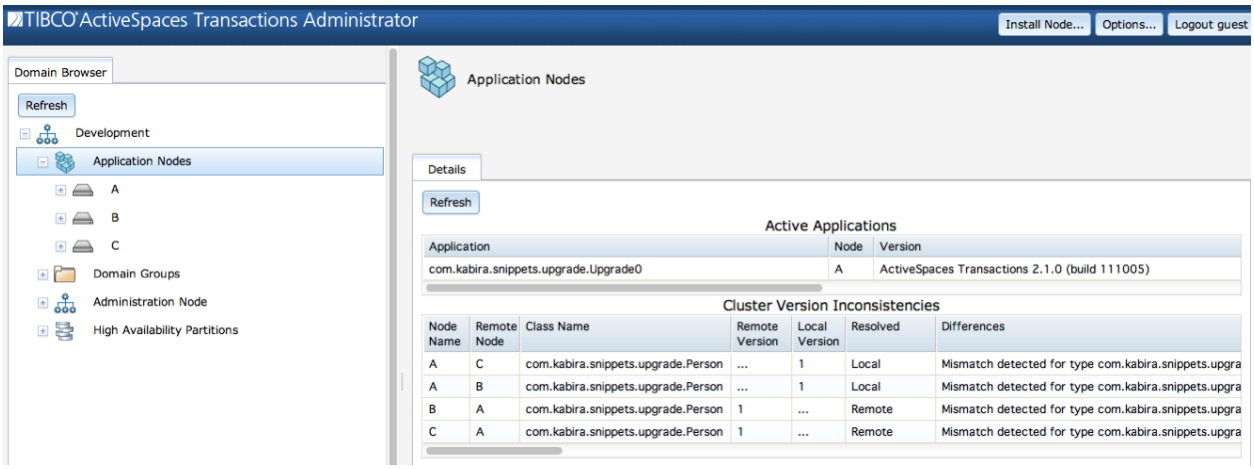

Figure 8.4. Cluster version mismatch

The object mapping can be seen by displaying the same object instance on two different nodes. Figure 8.5 shows the initial version of the `Person` object on replica node `B`. Notice that it only has two fields - `name` and `age`.

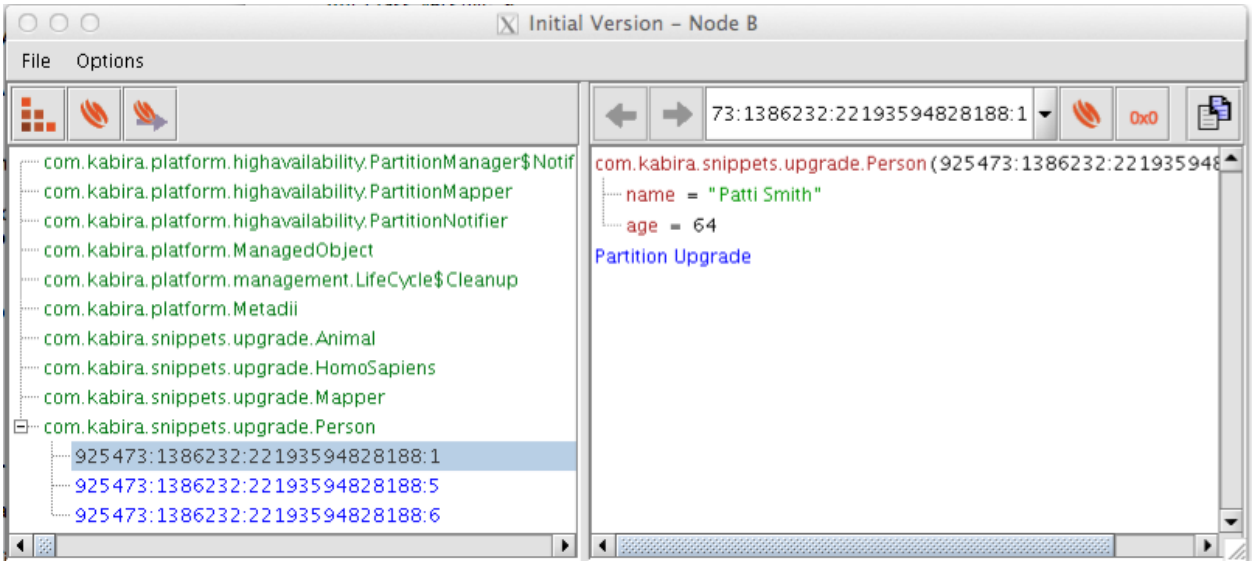

Figure 8.5. Initial version - node B

Compare the display of the same object on node `A`, which has been upgraded to the new class version. There was a new field, `type`, added by the new parent class which was initialized to a value of `Human`. The `name` field was also broke into two fields - `first` and `last`. Finally, the type of the `age` field was changed. It is important to emphasis that the different application versions on the two different nodes see the exact same object differently. The mapping is completely transparent.

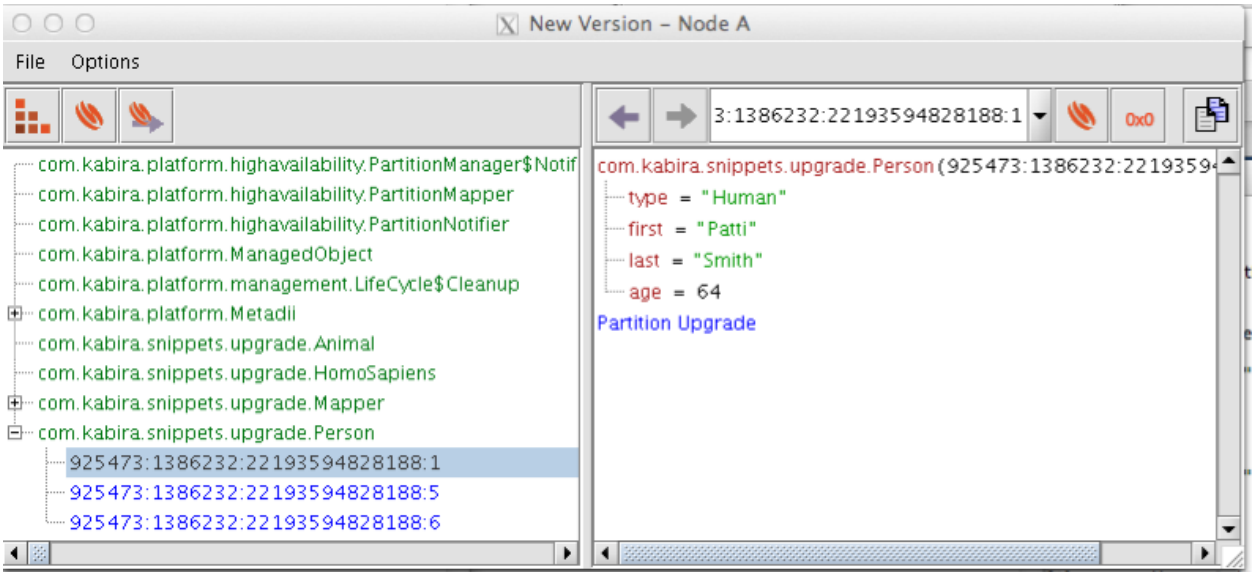

Figure 8.6. New version - node A

9

Configuration

This chapter describes how to define and use TIBCO ActiveSpaces® Transactions configuration.

Defining configuration objects

An TIBCO ActiveSpaces® Transactions configuration object is defined as a Java class that extends a known base type. A configuration notifier is also defined using inheritance.

Table 9.1. Configuration object definitions

Identification Mechanism	Description
`extends com.kabira.platform.kcs.Configuration`	A configuration object.
`extends com.kabira.platform.switchconfig.ConfigurationListener`	A configuration notifier.

Configuration objects and notifiers are managed objects - they can only be manipulated in a transaction. See Chapter 5 for more details on Managed Objects.

Here is an example of a configuration object.

Example 9.1. Configuration object

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.configuring;

import com.kabira.platform.kcs.Configuration;
import com.kabira.platform.annotation.Key;
import com.kabira.platform.annotation.KeyField;
import java.util.Date;

enum Gender
{
    MALE,
    FEMALE
```

```
}

/**
 * User configuration
 *
 * A key was added to allow the configuration data to be accessed
 * by last name. Notice that the versionId field also is in the
 * key.  This is required since there may be multiple versions
 * of this configuration data loaded, both with the same last name.
 */
@Key(name = "ByLastName",
fields =
{
    "lastName", "versionId"
},
unique = true,
ordered = false)
public class User extends Configuration
{

    User(
        @KeyField(fieldName = "lastName")
        final String lastName,
        @KeyField(fieldName = "versionId")
        final String version)
    {
        this.versionId = version;
        this.lastName = lastName;
    }
    String firstName;
    final String lastName;
    Long age;
    Gender gender;
    Date joined;

    //
    //  Phone number is an optional field
    //
    String phoneNumber = "Not provided";
}
```

See Example 9.9 on page 187 for an example of a configuration notifier.

Configuration type

By default the configuration type of a configuration object is the Java package name. All configuration objects defined in the same package will have the same configuration type For example, the configuration type of the configuration object defined in Example 9.1 on page 175 is `com.kabira.snippets.configuring`.

The default configuration type can be changed by overriding the `getType()` method. This method returns a string that is used for the configuration type.

For historical reasons the configuration type is called `GroupKind` in some parts of the configuration API. The documentation will always use *type* to describe this concept.

Example 9.2. Overriding default configuration type

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.configuring;

import com.kabira.platform.kcs.Configuration;
```

```
/**
 * A configuration object that overrides the default configuration type.
 *
 * This configuration object has a configuration type of MyConfigurationType
 */
public class TypeOverride extends Configuration
{
    @Override
    public String getType()
    {
        return "MyConfigurationType";
    }
    public String   value;
}
```

Optional fields

By default all fields specified in a configuration class are required in the configuration file. Adding a field initializer in a configuration object definition makes that field optional in the configuration file. If the field is not set in the configuration value, the configuration object will have the initializer value specified in the class definition. The `User.phoneNumber` field is optional in the `User` configuration object in Example 9.1 on page 175.

Supported field and array types

The supported types for fields and array elements in configuration objects are:

- `boolean and Boolean`
- `byte and Byte`
- `long and Long`
- `java.lang.String`
- `java.util.Date`
- Enumerations
- Managed Objects (including other configuration objects)
- Arrays containing one of the supported types.

Managed Objects and Array nesting is supported. See Example 9.3 on page 178 for an example of defining nested configuration. See Example 9.4 on page 178 for an example of how nested configuration data is specified in a configuration file.

An exception is thrown during configuration load if the configuration data contains unsupported types.

Fields in a configuration object can be either `public`, `package private`, or `private`.

Keys are supported, but not required, on configuration objects. See the section called “Keys and Queries” on page 59 for details on defining keys.

Configuration classes must be resolved by the JVM before configuration data can be loaded. See Example 9.7 on page 180 for details on one way to ensure that a configuration

class has been resolved by the JVM. If the configuration class has not been resolved, the configuration will fail to load with an unresolved class error.

Example 9.3. Nested configuration definition

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.configuring;

import com.kabira.platform.annotation.Managed;
import com.kabira.platform.kcs.Configuration;

@Managed
class Player
{
    String name;
    String position;
}

/**
 */
public class Team extends Configuration
{
    /**
     * Team name
     */
     String   name;

    /**
     * Team players
     */
     Player [] players;
}
```

Example 9.4. Nested configuration file

```
//     $Revision: 1.1.2.1 $
configuration "team" version "1.0" type "com.kabira.snippets.configuring"
{
        configure com.kabira.snippets.configuring
        {
                Team
                {
                        name = "giants";
                        players =
                        {
                            {
                                name = "Tim Lincecum";
                                position = "Pitcher";
                            },
                            {
                                name = "Matt Cain";
                                position = "Pitcher";
                            },
                            {
                                name = "Buster Posey";
                                position = "Catcher";
                            }
   };
                  };
        };
};
```

Data Format A Date value in a configuration file must be a string using a format of *Y-M-D T* where:

- `Y` is the year, including a century, expressed as a decimal number, e.g. `2011`.
- `M` is the month expressed as a decimal number from 01 to 12, e.g. `12` for December.
- `D` is the day of the month expressed as a decimal number from 01 to 31, e.g. `04`.
- `T` is the time expressed as `H:M:S` where `H` is a decimal number between representing a 24 hour clock from 00 to 23, `M` is the minute expressed as a decimal number from 00 to 59, and `S` is the second expressed as a decimal number from 00-60, e.g. `11:05:23`.

Configuration object life cycle

The creation and deletion of all configuration objects is done by the TIBCO ActiveSpaces® Transactions configuration framework. Any nested Managed Objects contained in a configuration object are also created and deleted by the configuration framework. They must not be deleted by the application.

Restrictions

Configuration objects have the following restrictions:

- all fields must be a supported Java type.
- all arrays must contain only supported Java types.
- nested Managed Objects cannot contain fields named `groupId` or `versionId`. This is because the configuration loader automatically populates these fields with the current configuration *name* (`groupId`) and *version* (`versionId`) when the data is loaded. These fields are removed from the configuration data when it is exported using the configuration exporter. If these fields are defined in nested Managed Objects, round-tripping of the configuration data will not work.

Accessing configuration objects

Configuration objects are accessed like any other Managed Object. They can be located using the extent or a query using unique or non-unique keys.

If a unique key is defined on a configuration object it must include the version identifier. This is required because different versions of a configuration object may be loaded at the same time. If the unique key did not include the version identifier a duplicate key exception would be reported when the second version of the configuration data was loaded. Example 9.1 on page 175 has an example of a unique key that contains the version identifier.

Example 9.7 on page 180 loads two versions of the same configuration data and then displays the configuration data using an extent query.

Example 9.5. User configuration version 1.0

```
//     $Revision: 1.1.2.1 $
configuration "user" version "1.0" type "com.kabira.snippets.configuring"
{
        configure com.kabira.snippets.configuring
        {
                User
                {
                        firstName = "John";
```

```
                        lastName = "Doe";
                        age = 8;
                        gender = Gender.MALE;
      joined = "2011-01-25 12:00:00";
                 };
                 User
                 {
                        firstName = "Big";
                        lastName = "Steve";
                        age = 19;
                        gender = Gender.MALE;
      joined = "2011-01-25 01:09:12";
    };
        };
};
```

Example 9.6. User configuration version 2.0

```
//     $Revision: 1.1.2.1 $
configuration "user" version "2.0" type "com.kabira.snippets.configuring"
{
        configure com.kabira.snippets.configuring
        {
                User
                {
                        firstName = "John";
                        lastName = "Doe";
                        age = 9;
                        gender = Gender.MALE;
       joined = "2011-01-25 08:04:00";
               };
                User
                {
                        firstName = "Big";
                        lastName = "Steve";
                        age = 87;
                        gender = Gender.MALE;
                        phoneNumber = "415-555-1212";
      joined = "2011-01-25 01:09:12";
    };
        };
};
```

Example 9.7. Locating configuration objects

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.configuring;

import com.kabira.test.management.Client;
import com.kabira.test.management.CommandFailed;

import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
import com.kabira.platform.switchconfig.Version;

import java.net.URL;

/**
 * Snippet to locate configuration objects by version
 *
 * NOTE: This snippet requires the classpath to include the
 * directory where the configuration files are located.  This
 * allows the get resource call to locate the configuration files
 *
 * <p>
 * <h2> Target Nodes</h2>
```

```
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class LocateConfiguration
{
    /**
     * Execute snippet
     * @param args                      Not used
     * @throws CommandFailed            Configuration command failed
     * @throws ClassNotFoundException   Could not load a class
     */
    public static void main(String[] args) throws CommandFailed, ClassNotFoundException
    {
        Client client = new Client("guest", "guest");

        URL url1 = client.getClass().getClassLoader().getResource("user1.kcs");
        URL url2 = client.getClass().getClassLoader().getResource("user2.kcs");
        URL url3 = client.getClass().getClassLoader().getResource("team.kcs");

        if ((url1 == null) || (url2 == null) || (url3 == null))
        {
            throw new Error(
                "Configuration files not located - are they installed into "
                + "the current class path?");
        }

        Client.Configuration version1 = client.new Configuration(url1);
        Client.Configuration version2 = client.new Configuration(url2);
        Client.Configuration version3 = client.new Configuration(url3);

        //
        //  Resolve configuration classes
        //
        Class.forName("com.kabira.snippets.configuring.User");
        Class.forName("com.kabira.snippets.configuring.Team");

        //
        //  Load configuration files
        //
        version1.load();
        version2.load();
        version3.load();

        //
        //  Activate version 1
        //
        version1.activate();
        displayData();

        //
        //  Activate version 2
        //
        version2.activate();
        displayData();

        //
        //  Deactive version 2
        //
        version2.deactivate();

        //
        //  Remove configurations
        //
        version1.remove();
        version2.remove();
        version3.remove();
```

```
    }

    //
    //  Display configuration data and versions
    //
    private static void displayData()
    {
        //
        //  Display the configuration objects
        //
        new Transaction("Locate Configuration")
        {
            @Override
            protected void run()
            {
                //
                //  Display user configuration objects
                //
                System.out.println("Configuration Data");
                for (User user : ManagedObject.extent(User.class))
                {
                    System.out.println(
                        "\tFirst Name: " + user.firstName
                        + " Last Name: " + user.lastName
                        + " Age: " + user.age
                        + " Joined: " + user.joined
                        + " Configuration Version: " + user.versionId
                        + " Configuration Name: " + user.groupId);
                }
                System.out.println("");
                for (Team team : ManagedObject.extent(Team.class))
                {
                    System.out.println(
                        "\tTeam Name: " + team.name
                        + " Configuration Version: " + team.versionId
                        + " Configuration Name: " + team.groupId);
                    for (Player player : team.players)
                    {
                        System.out.println(
                            "\t\tName: " + player.name
                            + " Position: " + player.position);
                    }
                }
                System.out.println("");

                //
                //  Display version data - skip all versions other
                //  then the ones associated with this configuration type
                //
                System.out.println("Versions");
                for (Version version : ManagedObject.extent(Version.class))
                {
                    if (version.groupKindId.equals(
                        "com.kabira.snippets.configuring") != true)
                    {
                        continue;
                    }
                    System.out.println(
                        "\tType: " + version.groupKindId
                        + " Name: " + version.groupId
                        + " Version: " + version.versionId
                        + " State: " + version.state.name());
                }
                System.out.println("");
            }
        }.execute();
```

```
    }
}
```

When this snippet is run the following output is displayed (annotation added).

```
#
#     All configuration data is loaded in memory
#
[A] Configuration Data
[A]   First Name: John Last Name: Doe Age: 9 Joined: Tue Jan 25 08:04:00 PST 2011
Configuration Version: 2.0 Configuration Name: user
[A]   First Name: Big Last Name: Steve Age: 87 Joined: Tue Jan 25 01:09:12 PST 2011
Configuration Version: 2.0 Configuration Name: user
[A]   First Name: John Last Name: Doe Age: 8 Joined: Tue Jan 25 12:00:00 PST 2011
Configuration Version: 1.0 Configuration Name: user
[A]   First Name: Big Last Name: Steve Age: 19 Joined: Tue Jan 25 01:09:12 PST 2011
Configuration Version: 1.0 Configuration Name: user
[A]
[A]   Team Name: giants Configuration Version: 1.0 Configuration Name: team
[A]     Name: Tim Lincecum Position: Pitcher
[A]     Name: Matt Cain Position: Pitcher
[A]     Name: Buster Posey Position: Catcher
[A]

#
#     Version 1.0 of the user configuration is active.  Version 2.0 of
#     the user configuration is inactive.  Version 1.0 of the team configuration
#     is inactive
#
[A] Versions
[A]   Type: com.kabira.snippets.configuring Name: user Version: 1.0 State: Active
[A]   Type: com.kabira.snippets.configuring Name: user Version: 2.0 State: Inactive
[A]   Type: com.kabira.snippets.configuring Name: team Version: 1.0 State: Inactive
[A]

#
#     All configuration data still in memory after activating version 2.0 of
#     user configuration
#
[A] Configuration Data
[A]   First Name: John Last Name: Doe Age: 9 Joined: Tue Jan 25 08:04:00 PST 2011
Configuration Version: 2.0 Configuration Name: user
[A]   First Name: Big Last Name: Steve Age: 87 Joined: Tue Jan 25 01:09:12 PST 2011
Configuration Version: 2.0 Configuration Name: user
[A]   First Name: John Last Name: Doe Age: 8 Joined: Tue Jan 25 12:00:00 PST 2011
Configuration Version: 1.0 Configuration Name: user
[A]   First Name: Big Last Name: Steve Age: 19 Joined: Tue Jan 25 01:09:12 PST 2011
Configuration Version: 1.0 Configuration Name: user
[A]
[A]   Team Name: giants Configuration Version: 1.0 Configuration Name: team
[A]     Name: Tim Lincecum Position: Pitcher
[A]     Name: Matt Cain Position: Pitcher
[A]     Name: Buster Posey Position: Catcher
[A]

#
#     Version 2.0 of the user configuration is active.  Version 1.0 of
#     the user configuration is inactive.  Version 1.0 of the team configuration
#     is inactive
#
[A] Versions
[A]   Type: com.kabira.snippets.configuring Name: user Version: 1.0 State: Inactive
[A]   Type: com.kabira.snippets.configuring Name: user Version: 2.0 State: Active
[A]   Type: com.kabira.snippets.configuring Name: team Version: 1.0 State: Inactive
```

Versions

To find an active version for a specific configuration *type* and *name* the `ByGroupState` key is used. This non-unique key allows `Version` objects to be selected using the configuration *type*, *name*, and *state*. The `ByGroupState` key is non-unique since there can be multiple `Inactive` versions for a specific configuration *type* and *name*. There is always only zero or one `Active` version.

For historical reasons the configuration *name* is called `Group` in the configuration API. The documentation will always use *name* to describe this concept.

All configuration objects associated with a version (active or not) are found using the `Version.getConfigs()` method. This is shown in Example 9.8 on page 184. Configuration objects are returned from the `Version.getConfigs()` method in the same order in which they were loaded from the configuration file.

Example 9.8. Version object

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.configuring;

import com.kabira.platform.KeyFieldValueList;
import com.kabira.platform.KeyManager;
import com.kabira.platform.KeyQuery;
import com.kabira.platform.LockMode;
import com.kabira.test.management.Client;
import com.kabira.test.management.CommandFailed;

import com.kabira.platform.Transaction;
import com.kabira.platform.switchconfig.Config;
import com.kabira.platform.switchconfig.Version;

import java.net.URL;

/**
 * Snippet to locate the active version
 *
 * NOTE: This snippet requires the classpath to include the
 * directory where the configuration files are located.  This
 * allows the get resource call to locate the configuration files
 *
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class ActiveVersion
{
    /**
     * Execute snippet
     * @param args      Not used
     * @throws CommandFailed            Configuration command filed
     * @throws ClassNotFoundException   Configuration class not found
     */
    public static void main(String[] args) throws CommandFailed, ClassNotFoundException
    {
        Client client = new Client("guest", "guest");

        URL url1 = client.getClass().getClassLoader().getResource("user1.kcs");
        URL url2 = client.getClass().getClassLoader().getResource("user2.kcs");

        if ((url1 == null) || (url2 == null))
```

```
        {
            throw new Error(
                "Configuration files not located - are they installed into "
                + "the current class path?");
        }

        //
        //  Resolve configuration classes
        //
        Class.forName("com.kabira.snippets.configuring.User");

        Client.Configuration version1 = client.new Configuration(url1);
        Client.Configuration version2 = client.new Configuration(url2);

        //
        //  Load configuration files
        //
        version1.load();
        version2.load();

        //
        //  Activate version 1
        //
        System.out.println("Activate version 1.0");
        version1.activate();
        displayActiveVersion();

        //
        //  Activate version 2
        //
        System.out.println("Activate version 2.0");
        version2.activate();
        displayActiveVersion();

        //
        //  Deactive version 2
        //
        version2.deactivate();

        //
        //  Remove configurations
        //
        version1.remove();
        version2.remove();
    }

    //
    //  Locate and display the active version
    //
    private static void displayActiveVersion()
    {
        new Transaction("Display Active Version")
        {
            @Override
            protected void run() throws Rollback
            {
                KeyQuery<Version> kq;
                KeyFieldValueList kfvl;
                //
                //  The active version is selected using the ByGroupState key.
                //
                kq = new KeyManager<Version>().
                    createKeyQuery(Version.class, "ByGroupState");
                kfvl = new KeyFieldValueList();
                kfvl.add("groupKindId", User.class.getPackage().getName());
                kfvl.add("groupId", "user");
                kfvl.add("state", Version.State.Active);
```

```
                kq.defineQuery(kfvl);

                for (Version version : kq.getResults(LockMode.READLOCK))
                {
                    System.out.println(
                        "Name: " + version.groupId
                        + " Type: " + version.groupKindId
                        + " Version: " + version.versionId
                        + " State: " + version.state.name());

                    Config[] configArray = version.getConfigs();
                    System.out.println("Contains:");
                    for (Config config : configArray)
                    {
                        User    user = (User)config;

                        if (user == null)
                        {
                            continue;
                        }

                        System.out.println(
                            "\tFirst Name: " + user.firstName
                            + " Last Name: " + user.lastName
                            + " Age: " + user.age
                            + " Gender: " + user.gender
                            + " Joined: " + user.joined
                            + " Phone Number: " + user.phoneNumber
                            + " Configuration Version: " + user.versionId
                            + " Configuration Name: " + user.groupId);
                    }
                }
            }
        }.execute();
    }
}
```

When this snippet is run it displays the following output (annotation added):

```
#
#    Version 1.0 is active
#
[A] Activate version 1.0
[A] Name: user Type: com.kabira.snippets.configuring Version: 1.0 State: Active
[A] Contains:
[A]  First Name: John Last Name: Doe Age: 8 Gender: MALE Joined: Tue Jan 25 12:00:00 PST
 2011 Phone Number: Not provided Configuration Version: 1.0 Configuration Name: user
[A]  First Name: Big Last Name: Steve Age: 19 Gender: MALE Joined: Tue Jan 25 01:09:12
PST 2011 Phone Number: Not provided Configuration Version: 1.0 Configuration Name: user

#
#     Version 2.0 is active
#
[A] Activate version 2.0
[A] Name: user Type: com.kabira.snippets.configuring Version: 2.0 State: Active
[A] Contains:
[A]  First Name: John Last Name: Doe Age: 9 Gender: MALE Joined: Tue Jan 25 08:04:00 PST
 2011 Phone Number: Not provided Configuration Version: 2.0 Configuration Name: user
[A]  First Name: Big Last Name: Steve Age: 87 Gender: MALE Joined: Tue Jan 25 01:09:12
PST 2011 Phone Number: 415-555-1212 Configuration Version: 2.0 Configu
```

Notifiers

Configuration notifiers provide a mechanism to perform application specific configuration auditing and to associate application behavior with configuration state changes.

Configuration notifiers are defined by extending `com.kabira.platform.switchconfig.ConfigurationListener`. Creating an instance of a user defined configuration notifier implicitly registers the notifier to be called by the configuration framework. Configuration notifiers can be created as part of application initialization and deleted as part of application termination. Another mechanism to install and remove configuration notifiers is to use TIBCO ActiveSpaces® Transactions component notifiers. See the section called "Notifiers" on page 223 for details.

There are two different kinds of methods in a configuration notifier:

- Audit
- State change

Audit methods are always called before its associated state change method (except for load where they are called after the load method). If a method throws the `com.kabira.platform.switchconfig.ConfigurationException` it is an audit method. An audit method reports an audit failure by throwing a `com.kabira.platform.switchconfig.ConfigurationException` exception. An audit failure prevents the requested configuration state change from occurring. Audit methods should not change the application state in any way.

A state change method cannot fail. It is only called if its associated audit method did not report an error. Any application state changes triggered by the configuration state change should be implemented in state methods.

The `loaded` state change method provides a mechanism to manually add additional configuration objects to the ones loaded from an external file. To add additional configuration objects, the implementation of the `loaded` method should create configuration instances and return them to the framework in the `additions` parameter.

Here is an example of a configuration notifier.

Example 9.9. Configuration notifier

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.configuring;

import com.kabira.platform.switchconfig.Config;
import com.kabira.platform.switchconfig.ConfigurationException;
import com.kabira.platform.switchconfig.ConfigurationListener;
import com.kabira.platform.switchconfig.Version;

/**
 * Configuration notifier
 */
public class UserNotifier extends ConfigurationListener
{

    public UserNotifier()
    {
        super( User.class.getPackage().getName());
    }

    @Override
    public void loaded(
        Version version, java.util.List<Config> additions)
    {
        System.out.println("Loading type: "
            + version.groupKindId
            + " name: "
            + version.groupId
```

```
            + " version: "
            + version.versionId);
    }

    @Override
    public void auditLoad(Version version) throws ConfigurationException
    {
        System.out.println("Auditing load for type: "
            + version.groupKindId
            + " name: "
            + version.groupId
            + " version: "
            + version.versionId);
    }

    @Override
    public void auditActivate(Version version) throws ConfigurationException
    {
        System.out.println("Auditing activation for type: "
            + version.groupKindId
            + " name: "
            + version.groupId
            + " version: "
            + version.versionId);
    }

    @Override
    public void activated(Version version)
    {
        System.out.println("Activating type: "
            + version.groupKindId
            + " name: "
            + version.groupId
            + " version: "
            + version.versionId);
    }

    @Override
    public void auditReplace(
        Version deactivating,
        Version activating) throws ConfigurationException
    {
        System.out.println("Auditing replace (deactivation) of type: "
            + deactivating.groupKindId
            + " name: "
            + deactivating.groupId
            + " version: "
            + deactivating.versionId);

        System.out.println("Auditing replace (activation) of type: "
            + activating.groupKindId
            + " name: "
            + activating.groupId
            + " version: "
            + activating.versionId);
    }

    @Override
    public void replaced(Version deactivating, Version activating)
    {
        System.out.println("Replacing (deactivation) type: "
            + deactivating.groupKindId
            + " name: "
            + deactivating.groupId
            + " version: "
            + deactivating.versionId);
```

```
        System.out.println("Replacing (activation) type: "
            + activating.groupKindId
            + " name: "
            + activating.groupId
            + " version: "
            + activating.versionId);
    }

    @Override
    public void auditDeactivate(Version version) throws ConfigurationException
    {
        System.out.println("Auditing deactivation of type: "
            + version.groupKindId
            + " name: "
            + version.groupId
            + " version: "
            + version.versionId);
    }

    @Override
    public void deactivated(Version version)
    {
        System.out.println("Deactivating type: "
            + version.groupKindId
            + " name: "
            + version.groupId
            + " version: "
            + version.versionId);
    }

    @Override
    public void auditRemove(Version version) throws ConfigurationException
    {
        System.out.println("Auditing removal of type: "
            + version.groupKindId
            + " name: "
            + version.groupId
            + " version: "
            + version.versionId);
    }

    @Override
    public void removed(Version version)
    {
        System.out.println("Removing type: "
            + version.groupKindId
            + " name: "
            + version.groupId
            + " version: "
            + version.versionId);
    }
}
```

Configuration notifiers are installed by creating an instance. They are removed by deleting the instance.

Example 9.10 on page 189 contains a complete example of initializing and terminating a configuration notifier. See Example 9.9 on page 187 for the notifier that is being installed.

Example 9.10. Notifier initialization and termination

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.configuring;

import com.kabira.platform.ManagedObject;
```

```
import com.kabira.platform.Transaction;
import com.kabira.test.management.Client;
import com.kabira.test.management.CommandFailed;
import java.net.URL;

/**
 * Snippet to install and terminate configuration notifiers
 *
 * NOTE: This snippet requires the classpath to include the
 * directory where the configuration files are located.  This
 * allows the get resource call to locate the configuration files
 *
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class NotifierLifeCycle
{
    /**
     * Initialize the configuration notifier
     */
    public void initialize()
    {
        new Transaction("Initialize Configuration Notifier")
        {
            @Override
            protected void run() throws Rollback
            {
                m_userNotifier = new UserNotifier();
            }
        }.execute();
    }

    /**
     * Terminate the configuration notifier
     */
    public void terminate()
    {
        new Transaction("Terminate Configuration Notifier")
        {
            @Override
            protected void run() throws Rollback
            {
                //
                //  Delete the configuration notifier
                //
                ManagedObject.delete(m_userNotifier);
                m_userNotifier = null;
            }
        }.execute();
    }

    /**
     * Execute snippet
     *
     * @param args              Not used
     * @throws CommandFailed    Configuration command failed
     * @throws ClassNotFoundException   Configuration class not found
     */
    public static void main(String[] args) throws CommandFailed, ClassNotFoundException
    {
        Client client = new Client("guest", "guest");

        URL url1 = client.getClass().getClassLoader().getResource("user1.kcs");
        URL url2 = client.getClass().getClassLoader().getResource("user2.kcs");
```

```
        if ((url1 == null) || (url2 == null))
        {
            throw new Error(
                "Configuration files not located - are they installed into "
                + "current class path?");
        }

        Client.Configuration version1 = client.new Configuration(url1);
        Client.Configuration version2 = client.new Configuration(url2);

        //
        //  Install configuration notifiers
        //
        NotifierLifeCycle   notifierLifeCycle = new NotifierLifeCycle();
        notifierLifeCycle.initialize();

        //
        //  Resolve configuration classes
        //
        Class.forName("com.kabira.snippets.configuring.User");

        //
        //  Load configuration files
        //
        version1.load();
        version2.load();

        //
        //  Activate version 1
        //
        version1.activate();

        //
        //  Activate version 2 - causes replace notifier to be called
        //
        version2.activate();


        //
        //  Deactive version 2
        //
        version2.deactivate();

        //
        //  Remove configurations
        //
        version1.remove();
        version2.remove();

        //
        //  Remove configuration notifiers
        //
        notifierLifeCycle.terminate();

    }
    private UserNotifier m_userNotifier;
}
```

When Example 9.10 on page 189 is run it outputs the following (annotation added):

```
#
#    Load and audit version 1.0
#
[A] Loading type: com.kabira.snippets.configuring name: user version: 1.0
[A] Auditing load for type: com.kabira.snippets.configuring name: user version: 1.0
```

```
#
#     Load and audit version 2.0
#
[A] Loading type: com.kabira.snippets.configuring name: user version: 2.0
[A] Auditing load for type: com.kabira.snippets.configuring name: user version: 2.0

#
#    Audit activation, and activate version 1.0
#
[A] Auditing activation for type: com.kabira.snippets.configuring name: user version:
1.0
[A] Activating type: com.kabira.snippets.configuring name: user version: 1.0

#
#    Audit replace, and then replace version 1.0 with version 2.0
#
[A] Auditing replace (deactivation) of type: com.kabira.snippets.configuring name: user
 version: 1.0
[A] Auditing replace (activation) of type: com.kabira.snippets.configuring name: user
version: 2.0
[A] Replacing (deactivation) type: com.kabira.snippets.configuring name: user version:
1.0
[A] Replacing (activation) type: com.kabira.snippets.configuring name: user version: 2.0

#
#     Audit deactivation, and then deactivate version 2.0
#
[A] Auditing deactivation of type: com.kabira.snippets.configuring name: user version:
2.0
[A] Deactivating type: com.kabira.snippets.configuring name: user version: 2.0

#
#    Audit removal, and then remove, version 1.0
#
[A] Auditing removal of type: com.kabira.snippets.configuring name: user version: 1.0
[A] Removing type: com.kabira.snippets.configuring name: user version: 1.0

#
#    Audit removal, and then remove, version 2.0
#
[A] Auditing removal of type: com.kabira.snippets.configuring name: user version: 2.0
[A] Removing type: com.kabira.snippets.configuring name: user version: 2.0
```

Defining partitions using configuration

Example 9.11 on page 192 shows an example of using configuration to define and enable highly available partitions (see the section called “Defining and enabling partitions” on page 114). In this snippet, if a partition in the configuration data defines the active node as the local node, the partition is defined and enabled.

The snippet loads and activates the configuration file from `main`. However, in general, an application would install a configuration notifier, and the configuration data would be automatically loaded as part of node startup, or by the operator during normal operation.

Example 9.11. Partition definition in configuration

```
//     $Revision: 1.1.2.3 $
package com.kabira.snippets.configuring;

import com.kabira.platform.highavailability.ReplicaNode;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
```

```
import com.kabira.platform.Transaction.Rollback;
import com.kabira.test.management.Client;
import com.kabira.test.management.CommandFailed;
import java.net.URL;
import com.kabira.platform.kcs.Configuration;
import com.kabira.platform.switchconfig.Config;
import com.kabira.platform.switchconfig.ConfigurationListener;
import com.kabira.platform.switchconfig.Version;
import com.kabira.platform.highavailability.PartitionManager;
import com.kabira.platform.highavailability.Partition.Properties;
import static com.kabira.platform.highavailability.PartitionManager.EnableAction.*;
import static com.kabira.platform.highavailability.ReplicaNode.ReplicationType;
import com.kabira.platform.property.Status;

/**
 *  Partition configuration object
 */

@Managed
class Replica
{
    String name;
    ReplicationType type;
}

/**
 * This snippet demonstrates the use of configuration for defining partitions.
 *
 * NOTE: This snippet requires the classpath to include the
 * directory where the configuration files are located.  This
 * allows the get resource call to locate the configuration files
 *
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainname</b> = A
 * </ul>
 */
public class Partition extends Configuration
{
    String  name;
    String activeNode;
    String restoreFromNode = "";
    Boolean forceReplication = false;
    Replica [] replicas;

    /**
     * Set configuration type
     * @return Configuration type
     */
    @Override
    public String getType()
    {
        return CONFIGURATION_TYPE;
    }

    //
    //  Configuration notifier
    //
    private static class Notifier extends ConfigurationListener
    {
        static final String localNode = System.getProperty(Status.NODE_NAME);

        Notifier()
        {
            super(CONFIGURATION_TYPE);
        }
```

```
    @Override
    public void activated(Version version)
    {
        Boolean needEnable = false;

        for (Config config : version.getConfigs())
        {
            Partition   partition = (Partition)config;
            //
            //  Skip partitions that are not active on this node
            //
            if (partition.activeNode.equals(localNode) == false)
            {
                continue;
            }
            needEnable = true;

            //
            //  Define the partition properties
            //
            Properties  properties = new Properties();
            properties.forceReplication(partition.forceReplication);
            properties.restoreFromNode(partition.restoreFromNode);

            //
            //  Define the partition
            //
            ReplicaNode [] replicas = new ReplicaNode[partition.replicas.length];

            for (int i = 0; i < partition.replicas.length; i++)
            {
                replicas[i] = new ReplicaNode(
                    partition.replicas[i].name,
                    partition.replicas[i].type);
            }
            PartitionManager.definePartition(
                partition.name, properties, partition.activeNode, replicas);
        }

        //
        //  Enable the partitions
        //
        if (needEnable == true)
        {
            PartitionManager.enablePartitions(JOIN_CLUSTER);
        }
    }
}


public static void main(String[] args) throws CommandFailed, ClassNotFoundException
{
    Client client = new Client("guest", "guest");

    URL url1 = client.getClass().getClassLoader().getResource("partitions.kcs");

    if (url1 == null)
    {
        throw new Error(
            "Configuration file partitions.kcs not located - "
            + "is it installed into the current class path?");
    }

    Client.Configuration version = client.new Configuration(url1);

    //
```

```
        //  Install the configuration notifier
        //
        new Transaction("Install configuration notifier")
        {
            @Override
            protected void run() throws Rollback
            {
                new Notifier();
            }
        }.execute();

        //
        //  Install configuration notifiers
        //
        NotifierLifeCycle   notifierLifeCycle = new NotifierLifeCycle();
        notifierLifeCycle.initialize();


        //
        //  Load, activate configuration file
        //
        version.load();
        version.activate();
        version.deactivate();
        version.remove();

        //
        //  Remove the configuration notifier
        //
        new Transaction("Remove configuration notifier")
        {
            @Override
            protected void run() throws Rollback
            {
                for (Object notifier : ManagedObject.extent(Notifier.class))
                {
                    ManagedObject.delete(notifier);
                }
            }
        }.execute();

    }
    private static final String CONFIGURATION_TYPE = "partition";
}
```

Example 9.12 on page 195 shows the configuration data captured for each partition. Partition configuration includes:

- `name` - partition name
- `nodeList` - list of nodes associated with the partition. The first node in the list is the active node for the partition.

Example 9.12. Partition configuration

```
//     $Revision: 1.1.2.1 $
configuration "partitions" version "1.0" type "partition"
{
    configure com.kabira.snippets.configuring
    {
        //
        //    Define partition A with an active node A
        //
        Partition
        {
```

```
            name = "A";
            activeNode = "A";
        restoreFromNode = "B";
        forceReplication = true;
        replicas =
        {
            {
                name = "B";
                type = SYNCHRONOUS;
            },
            {
                name = "C";
                type = ASYNCHRONOUS;
            }
        };
    };

    //
    //    Define partition B with an active node B
    //
    Partition
    {
            name = "B";
            activeNode = "B";
        replicas =
        {
            {
                name = "A";
                type = SYNCHRONOUS;
            },
            {
                name = "C";
                type = ASYNCHRONOUS;
            }
        };
    };
};
```

After this snippet has been run on node A and B, displaying the defined partitions in *TIBCO ActiveSpaces® Transactions* shows the information in Figure 9.1.

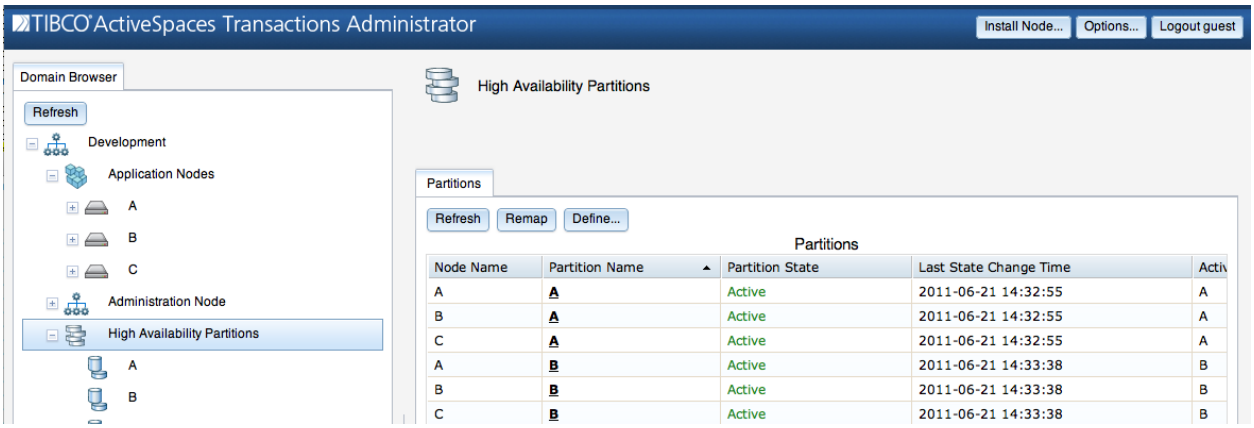

Figure 9.1. Configured partitions

Runtime objects

The purpose of configuration objects is to provide data for runtime objects. A runtime object is an application specific object that contains both state and behavior that provides an application specific function. This section describes common patterns for runtime objects and some guidelines for managing them.

Design patterns

There are various patterns you can use to define the correspondence between configuration and runtime objects; the following sections discuss some of these patterns.

Matching configuration and runtime objects In many cases, each configuration object is used to create and configure a runtime counterpart. The runtime object can be a Managed Object or a POJO. For example, when a Printer configuration object is activated, the activate notifier method would create a runtime printer object. The configuration data is copied into the runtime object fields.

Similarly, when the configuration object is deactivated, the runtime object is deleted or released. This is the least complex pattern to use, and results in the fewest design and implementation issues.

This pattern is best used when the application quickly locates a runtime object, uses it, and detaches. This pattern is also used when it seems that the existence of the runtime object should be initiated by activating a configuration, and destroyed by deactivating a configuration.

Differing life cycles Another pattern is used when a small number of runtime objects have a lifecycle that must be longer than the configuration object. In this case, upon activation, the configuration object should simply locate the runtime object, copy attribute values from the configuration object to the runtime object, and notify the runtime object to take appropriate action based on the configuration change. This pattern is also simple to use.

Shared configuration objects Another pattern is used when a large number of runtime objects share the same configuration objects. A large number could be thousands, or even millions of runtime objects. This pattern is used when the configuration objects form a repository, as a product catalog might for an example application. The product catalog (configuration data) defines the characteristics of the product, while the runtime objects would represent instances of products. In this case, the runtime objects would attach to the configuration object every time it needs to use the configuration data.

Runtime objects should never cache handles to configuration objects. These handles will become stale when the configuration version state changes.

10

Secondary Stores

This chapter describes how to transparently integrate managed objects with a secondary store such as an RDBMS, files, or long-term archival storage. The secondary store API defines a set of notifiers that are triggered as applications perform actions against managed objects such as creating, deleting, modifying, or querying. Applications use the normal managed object APIs - they are not aware that the managed objects are being externalized or read from a secondary store.

While the primary use of the secondary store notifiers is to integrate with external stores, there is nothing preventing them from being used for other functions such as auditing, change notification, etc.

The secondary store notifiers are defined in the `com.kabira.store` package. These abstract classes define the supported notifiers:

- `Extent<T>` - called when `ManagedObject.extent(...)` is executed.
- `Query<T>` - called when a query is performed.
- `Record<T>` - called when a managed object is created, modified, or deleted.

There is no requirement to implement all notifiers. Only the notifiers required to support the desired functionality must be implemented. Secondary store notifiers must be installed on all nodes that are interested in receiving notifications.

Notifiers are always called in the same transaction in which the application action was performed that triggered the notifier.

Distributed queries cause query notifiers, `Extent<T>` or `Query<T>`, to be executed on all nodes that are part of the distributed query. The notifiers must be installed on all remote nodes that are involved in the distributed query.

Object modifications cause `Record<T>` notifiers to be executed on the current active node for the object being modified. The notifier must be installed on the active node to be executed.

The Extent<T> and Query<T> notifiers make objects available to the application query by creating, updating, or deleting managed objects as required to satisfy the application request. For example, an Extent<T> notifier that integrates with an external RDBMS might perform a `select * from` <T> query against the database and create objects for all returned rows.

Managed object creates, deletes, modifications, or queries, done by the implementation of a notifier do not cause any notifier methods to be triggered. Secondary store notifiers are only triggered by application actions. The only exception to this rule is if a notifier implementation performs a distributed query. In that case any installed query notifiers, Extent<T> or Query<T>, on remote nodes involved in the distributed query will be executed.

Lifecycle

Secondary store notifiers can be dynamically installed and removed. They are installed by creating an instance of a notifier. Any notifier previously installed for the type are removed. See the section called “Chaining notifiers” on page 217 for details on notifier chaining.

Notifiers are removed by deleting the notifier instance. They are also automatically removed when the JVM in which they were created is stopped. Notifiers must be reinstalled when the JVM is restarted.

Example 10.1. Secondary store notifier life-cycle

```
//     $Revision: 1.1.2.2 $
package com.kabira.snippets.store;

import com.kabira.platform.LockMode;
import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;
import com.kabira.store.Extent;

/**
 * Secondary store notifier life-cycle
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class LifeCycle
{
    @Managed
    private static class A { };

    private static class ExtentNotifier extends Extent<A>
    {
        public ExtentNotifier()
        {
            super(A.class);
        }

        @Override
        public void extent(Class<? extends A> type, LockMode lockMode)
        {
            // do something interesting here
        }
    }

    /**
     * Main entry point
```

```
     * @param args  Not used
     */
    public static void main(final String [ ] args)
    {
        //
        //  Install an extent notifier for managed class A
        //
        new Transaction("Install")
        {
            @Override
            protected void run() throws Transaction.Rollback
            {
                m_notifier = new ExtentNotifier();
            }
        }.execute();

        //
        //  Remove the extent notifier for managed class A
        //
        new Transaction("Remove")
        {
            @Override
            protected void run() throws Transaction.Rollback
            {
                ManagedObject.delete(m_notifier);
                m_notifier = null;
            }
        }.execute();

        //
        //  Re-install an extent notifier for managed class A
        //  This notifier will be removed when the JVM is stopped
        //
        new Transaction("Re-Install")
        {
            @Override
            protected void run() throws Transaction.Rollback
            {
                m_notifier = new ExtentNotifier();
            }
        }.execute();
    }

    private static ExtentNotifier m_notifier;
}
```

Transaction management

Secondary store notifiers are responsible for managing transactions, or simulating transactional behavior, to external secondary stores. Any errors detected when communicating with a secondary store should be handled in the notifier and the current transaction rolled back to restore shared memory state to its value before the transaction was started. If the transaction is not rolled back the state in shared memory is out of sync with the external secondary store.

Secondary store notifiers should define and create a transaction notifier to manage transactional interaction with a secondary store. The transaction notifier should be created the first time a notifier is called. In general, only a single transaction notifier should be created per transaction. For example, if the external secondary store is an RDBMS, the transaction notifier should be allocated when a connection to the data base is allocated.

The Example 10.2 on page 202 demonstrates creating a transaction notifier and simulating an error when communciating with an external secondary store.

Example 10.2. Transaction management

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.store;

import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;
import com.kabira.platform.swbuiltin.TransactionNotifier;
import com.kabira.store.Record;

/**
 * Secondary store transaction management
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class TransactionManagement
{
    @Managed
    private static class A
    {
        A(boolean error)
        {
            this.error = error;
        }
        final boolean error;
    };

    private static class RecordNotifier extends Record<A>
    {
        RecordNotifier()
        {
            super(A.class);
        }

        @Override
        public void created(A a)
        {
            System.out.println("INFO: Object created");

            //
            //    Create transaction notifier to transactionally manage
            //    external store
            //
            new TransactionBoundary();

            //
            //    Simulate an error.  Throw an exception to rollback the
            //    transaction so that shared memory state is restored to
            //    its value before the transaction was started.
            //
            if (a.error == true)
            {
                System.out.println("INFO:\tRolling back transaction");
                throw new RuntimeException("Simulating external store error");
            }
        }
    }

    private static class TransactionBoundary extends TransactionNotifier
    {

        @Override
        public void onRollback()
        {
```

```
            System.out.println("INFO: Transaction "
                + Transaction.getIdentifier()
                + " rolled back.");

            //
            //    Undo any work in an external store
            //
        }

        @Override
        public void onCommit()
        {
            System.out.println("INFO: Transaction "
                + Transaction.getIdentifier()
                + " committed.");

            //
            //    Commit any work in an external store
            //    Commits cannot fail
            //
        }

        @Override
        public void onPrepare()
        {
            System.out.println("INFO: Transaction "
                + Transaction.getIdentifier()
                + " prepared.");

            //
            //    Prepare any work in an external store
            //
        }
    }

    /**
     * Main entry point
     *
     * @param args Not used
     */
    public static void main(final String[] args)
    {
        //
        //    Initialize record notifier
        //
        new Transaction("Initialize")
        {
            @Override
            protected void run()
            {
                new RecordNotifier();
            }
        }.execute();

        //
        //    Create an object with no error
        //
        new Transaction("Create object")
        {
            @Override
            protected void run()
            {
                new A(false);
            }
        }.execute();

        //
```

```
        //    Create an object with an error
        //
        new Transaction("Create object with error")
        {
            @Override
            protected void run()
            {
                new A(true);
            }
        }.execute();
    }
}
```

When Example 10.2 on page 202 is run it outputs these messages (annotation added):

```
#
#  created method called
#
INFO: Object created

#
#  Prepare and commit transaction notifier methods called
#
INFO: Transaction 129:2 prepared.
INFO: Transaction 129:2 committed.

#
#  created method called
#
INFO: Object created

#
#  Simulate error to external secondary store
#
INFO:    Rolling back transaction

#
#  Transaction rolled back
#
INFO: Transaction 129:3 rolled back.
Java main class com.kabira.snippets.store.TransactionManagement.main exited with an
exception.
com.kabira.platform.ResourceUnavailableException: java.lang.RuntimeException: Simulating
 external store error
    at
com.kabira.snippets.store.TransactionManagement$RecordNotifier.$createdImpl(TransactionManagement.java:55)

    at
com.kabira.snippets.store.TransactionManagement$RecordNotifier.created(TransactionManagement.java)

    at
com.kabira.snippets.store.TransactionManagement$RecordNotifier.created(TransactionManagement.java:29)

    at com.kabira.store.Record.$modifiedInternalImpl(Record.java:151)
    at com.kabira.store.Record.modifiedInternal(Record.java)
    at com.kabira.platform.Transaction.prepare(Native Method)
    at com.kabira.platform.Transaction.execute(Transaction.java:485)
    at com.kabira.platform.Transaction.execute(Transaction.java:542)
   at com.kabira.snippets.store.TransactionManagement.main(TransactionManagement.java:142)
```

Extent notifier

The `extent` method in `Extent<T>` notifiers is called when the application calls `ManagedObject.extent(...)`. The `extent` method is called on all nodes that are part of a distributed

query on which `Extent<T>` notifiers are installed. If a notifier is not installed on a node participating in a distributed query, the query succeeds without calling a notifier on that node.

The `extent` method is executed synchronously before the application performs the first iteration on the returned extent `Iterator`. The extent notifier controls the objects that will be returned in the extent iterator by creating and deleting managed objects as required.

Extent queries against large data sets can cause significant resource consumption, both memory and CPU, as managed objects are created and updated. Care should be taken to not negatively impact application performance during extent queries.

When an `Extent<T>` notifier is installed it is also installed on all child types, unless there is already a notifier installed on the child types. When an extent iteration is performed on a type, the `extent` method is called on all children first, starting at the most distant child and traversing up the inheritance hierarchy. In the case where an `Extent<T>` notifier is only installed on a base type, the `extent` method will be called once for each child in the type hierarchy. The `klass` parameter to the `extent` method can be used to determine the specific type on which the `extent` method is currently being called.

The application requested `LockMode` is in the `lockMode` parameter. The notifier should attempt to honor the requested `lockMode`, but there is no enforcement.

Example 10.3. Extent notifier

```
//     $Revision: 1.1.2.3 $
package com.kabira.snippets.store;

import com.kabira.platform.LockMode;
import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;
import com.kabira.store.Extent;

/**
 * Secondary store extent notifier
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class ExtentNotifier
{
    @Managed
    private static class Parent { };
    private static class Child extends Parent { };

    private static class Notifier extends Extent<Parent>
    {
        public Notifier()
        {
            super(Parent.class);
        }

        @Override
        public void extent(Class<? extends Parent> type, LockMode lockMode)
        {
            //
            //  Extent query on parent
            //
            if (type.equals(Parent.class) == true)
            {
```

```
                System.out.println("INFO: Extent called on parent");

                //
                //  Create some parent objects to return to application
                //
                new Parent();
                new Parent();
            }
            //
            //  Extent query on child
            //
            else
            {
                assert type.equals(Child.class) : type;

                System.out.println("INFO: Extent called on child");

                //
                //  Create some child objects to return to application
                //
                new Child();
                new Child();
            }
        }
    }

    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(final String [ ] args)
    {
        //
        //  Install extent notifier for Parent type
        //
        new Transaction("Install")
        {
            @Override
            protected void run() throws Transaction.Rollback
            {
                new Notifier();
            }
        }.execute();

        new Transaction("Extent Query")
        {
            @Override
            protected void run() throws Transaction.Rollback
            {
                //
                //  Perform extent query on child
                //
                for (Child c : ManagedObject.extent(Child.class))
                {
                    System.out.println("INFO:\t" + c);
                }

                //
                //  Perform extent query on parent
                //
                for (Parent p : ManagedObject.extent(Parent.class))
                {
                    System.out.println("INFO:\t" + p);
                }
            }
        }.execute();
```

```
    }
}
```

When the Example 10.3 on page 205 is run it outputs these messages (annotation added):

```
#
# Extent query performed on Child
#
INFO: Extent called on child
INFO:    com.kabira.snippets.store.ExtentNotifier$Child@865069a6
INFO:    com.kabira.snippets.store.ExtentNotifier$Child@25d51d22

#
# Extent query performed on Parent.  extent method called
# twice - once for Child and once for Parent
#
INFO: Extent called on child
INFO: Extent called on parent
INFO:    com.kabira.snippets.store.ExtentNotifier$Parent@4c936ce7
INFO:    com.kabira.snippets.store.ExtentNotifier$Parent@ec182063
INFO:    com.kabira.snippets.store.ExtentNotifier$Child@1ae3c08c
INFO:    com.kabira.snippets.store.ExtentNotifier$Child@ba687408
INFO:    com.kabira.snippets.store.ExtentNotifier$Child@865069a6
INFO:    com.kabira.snippets.store.ExtentNotifier$Child@25d51d22
```

Record notifier

The `Record<T>` notifier provides notification for object creation, modification, and deletion. Specifically, these methods:

- `created` - called when an object is created.
- `modified` - called when an object is modified.
- `deleted` - called when an object is deleted.

The methods in this notifier are always called on the master node for an object.

The `created` and `deleted` methods are not called if an object is created and deleted in the same transaction.

The `modified` method is only called once per transaction, even if an object is modified multiple times. The `modified` method is not called for objects that are created or deleted in a transaction.

The `deleted` method is called before an object is deleted.

`Record<T>` notifiers are inherited by any children of the type on which they are installed, unless a `Record<T>` notifier is explicitly installed on one of the child types.

Example 10.4. Record notifier

```
//     $Revision: 1.1.2.3 $
package com.kabira.snippets.store;

import com.kabira.platform.ManagedObject;
import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;
import com.kabira.store.Record;

/**
 * Secondary store record notifier
```

```
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class RecordNotifier
{
    @Managed
    private static class Parent
    {
        void increment()
        {
            m_count++;
        }
        private long m_count = 0;
    };
    private static class Child extends Parent { };

    private static class Notifier extends Record<Parent>
    {
        public Notifier()
        {
            super(Parent.class);
        }

        @Override
        public void deleted(Parent parent)
        {
            System.out.println("INFO: Deleted " + parent);
        }

        @Override
        public void modified(Parent parent)
        {
            System.out.println("INFO: Modified " + parent);

        }

        @Override
        public void created(Parent parent)
        {
            System.out.println("INFO: Created " + parent);

        }
    }

   /**
     * Main entry point
     * @param args  Not used
     */
    public static void main (final String [ ] args)
    {
        new Transaction("Create Child")
        {
            @Override
            protected void run()
            {
                new Notifier();
                m_child = new Child();
            }
        }.execute();

        new Transaction("Modify Child")
        {
            @Override
            protected void run()
```

```
            {
                m_child.increment();
            }
        }.execute();

        new Transaction("Delete Child")
        {
            @Override
            protected void run()
            {
                ManagedObject.delete(m_child);
                m_child = null;
            }
        }.execute();
    }

    private static Child m_child;
}
```

When the Example 10.4 on page 207 is run it outputs these messages:

```
INFO: Created com.kabira.snippets.store.RecordNotifier$Child@106c202d
INFO: Modified com.kabira.snippets.store.RecordNotifier$Child@106c202d
INFO: Deleted com.kabira.snippets.store.RecordNotifier$Child@106c202d
```

Query notifier

The `Query<T>` notifier provides notification for queries initiated by an application using the `KeyQuery<T>` class. The methods in this notifier are called on all nodes that are part of a distributed query on which `Query<T>` notifiers are installed. If a notifier is not installed on a node participating in a distributed query, the query succeeds without calling a notifier on that node.

Query notifier methods are always called, even if an application query would have been satisified by managed objcts in memory. The query notifier methods are responsible for determining whether a query should be passed to a secondary store or not based on the current contents of memory.

These methods are executed synchronously before the `KeyQuery<T>` API calls return to the application. The methods in this notifier control the query result set by creating and deleting managed objects as required.

The methods supported are:

- `query` - called for `KeyQuery<T>.getOrCreateSingleResult(...)`, `KeyQuery<T>.getResults(...)`, and `KeyQuery<T>.getSingleResult(...)`.
- `queryMaximum` - called for `KeyQuery<T>.getMaximumResult(...)`.
- `queryMinimum` - called for `KeyQuery<T>.getMinimumResult(...)`.

`Query<T>` notifiers are installed for a specific key. They must be installed on the type that defines the key. They are inherited by all children of the type on which they are installed. The `klass` parameter to the methods can be used to determine the actual type on which the query is being performed.

The `queryData` parameter provides details on the application specified query information. It is used to reconstruct the query in the notifier.

The application requested `LockMode` is in the `lockMode` parameter. The notifier should attempt to honor the requested `lockMode`, but there is no enforcement.

Result set ordering is the responsibility of the TIBCO ActiveSpaces® Transactions runtime, not the query notifiers. Any ordering specification on the application query is not available to the query notifiers as that information is not needed.

The Example 10.5 on page 210 shows how the application specified query is reconstructed in the query notifier methods.

Example 10.5. Query notifier

```
//     $Revision: 1.1.2.4 $
package com.kabira.snippets.store;

import com.kabira.platform.KeyComparisonOperator;
import com.kabira.platform.KeyFieldValueList;
import com.kabira.platform.KeyFieldValueRangeList;
import com.kabira.platform.KeyManager;
import com.kabira.platform.KeyOrderedBy;
import com.kabira.platform.KeyQuery;
import com.kabira.platform.LockMode;
import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Key;
import com.kabira.platform.annotation.Managed;
import com.kabira.store.KeyField;
import com.kabira.store.Query;
import java.util.ArrayList;

/**
 * Secondary store query notifier
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class QueryNotifier
{
    @Managed
    @Key(name = "ByName", unique = true, ordered = true, fields =
    {
        "m_name"
    })
    private static class A
    {
        private A(final String name)
        {
            m_name = name;
        }

        private final String m_name;
    };

    private static class B extends A
    {
        public B(final String name)
        {
            super(name);
        }
    };

    //
    //  Query notifier
    //
```

```
    private static class Notifier extends Query<A>
    {
        public Notifier(final String keyName)
        {
            super(A.class, keyName);
        }

        @Override
        public void query(
            Class<? extends A> klass,
            LockMode lockMode,
            ArrayList<KeyField> queryData)
        {
            System.out.println("INFO:\t" + prettyPrintQuery(
                "object",
                klass.getSimpleName(),
                getKeyName(),
                lockMode,
                queryData));
        }

        @Override
        public void queryMinimum(
            Class<? extends A> klass,
            LockMode lockMode,
            ArrayList<KeyField> queryData)
        {
            System.out.println("INFO:\t" + prettyPrintQuery(
                "minimum",
                klass.getSimpleName(),
                getKeyName(),
                lockMode,
                queryData));
        }

        @Override
        public void queryMaximum(
            Class<? extends A> klass,
            LockMode lockMode,
            ArrayList<KeyField> queryData)
        {
            System.out.println("INFO:\t" + prettyPrintQuery(
                "maximum",
                klass.getSimpleName(),
                getKeyName(),
                lockMode,
                queryData));
        }
    }

/**
 * Main entry point
 * @param args Not used
 */
    public static void main(final String[] args)
    {
        initialize();
        uniqueQuery();
        minimumMaximumQuery();
        rangeQuery();
    }

    //
    //    Initialize notifier and create some test data
    //
    private static void initialize()
    {
```

```
        new Transaction("Initialize")
        {
            @Override
            protected void run() throws Transaction.Rollback
            {
                new Notifier("ByName");

                for (int i = 0; i < 20; i++)
                {
                    new A(Integer.toString(i));
                }
                for (int i = 20; i < 30; i++)
                {
                    new B(Integer.toString(i));
                }
            }
        }.execute();
    }

    //
    //    Perform a unique query
    //
    private static void uniqueQuery()
    {
        new Transaction("Unique Query")
        {
            @Override
            protected void run() throws Transaction.Rollback
            {
                KeyManager<B> km = new KeyManager<>();
                KeyQuery<B> kq = km.createKeyQuery(B.class, "ByName");
                KeyFieldValueList values = new KeyFieldValueList();
                values.add("m_name", "22");
                kq.defineQuery(values);

                System.out.println("INFO: getSingleResult " + kq);
                B b = kq.getSingleResult(LockMode.NOLOCK);
            }
        }.execute();
    }

    //
    //    Perform a range query
    //
    private static void rangeQuery()
    {
        new Transaction("Range Query")
        {
            @Override
            protected void run() throws Transaction.Rollback
            {
                KeyManager<B> km = new KeyManager<>();
                KeyQuery<B> kq = km.createKeyQuery(B.class, "ByName");
                KeyFieldValueRangeList values = new KeyFieldValueRangeList();
                values.add("m_name", "22", KeyComparisonOperator.GT);
                values.add("m_name", "28", KeyComparisonOperator.LTE);
                kq.defineQuery(values);

                System.out.println("INFO: Range Query " + kq);

                for (B b : kq.getResults(KeyOrderedBy.ASCENDING, LockMode.NOLOCK))
                {
                    break;
                }
            }
        }.execute();
    }
```

```
//
//    Perform minimum and maximum queries
//
private static void minimumMaximumQuery()
{
    new Transaction("Minimum/Maximum Query")
    {
        @Override
        protected void run() throws Transaction.Rollback
        {
            KeyManager<A> km = new KeyManager<>();
            KeyQuery<A> kq = km.createKeyQuery(A.class, "ByName");

            System.out.println("INFO: Get Maximum Result");

            A a = kq.getMaximumResult(LockMode.READLOCK);

            KeyFieldValueRangeList values = new KeyFieldValueRangeList();
            values.add("m_name", "8", KeyComparisonOperator.GTE);
            kq.defineQuery(values);

            System.out.println("INFO: Get Minimum Result " + kq);
            a = kq.getMinimumResult(LockMode.NOLOCK);
        }
    }.execute();
}

//
//  Pretty print query
//
private static String prettyPrintQuery(
    final String queryType,
    final String className,
    final String keyName,
    LockMode lockMode,
    ArrayList<KeyField> queryData)
{
    StringBuilder builder = new StringBuilder();

    builder.append("select ");
    builder.append(queryType);
    builder.append(" from ");
    builder.append(className);
    builder.append(" using ");
    builder.append(keyName);
    builder.append(prettyPrintQueryData(queryData));
    builder.append(" with ");
    builder.append(lockMode);

    return builder.toString();
}

//
//  Pretty print query data
//
private static String prettyPrintQueryData(final ArrayList<KeyField> queryData)
{
    if (queryData.isEmpty() == true)
    {
        return "";
    }

    StringBuilder builder = new StringBuilder();
    boolean first = true;

    builder.append(" where (");
```

```
        for (KeyField f : queryData)
        {
            if (first == true)
            {
                first = false;
            }
            else
            {
                builder.append(" && ");
            }
            builder.append(f.name);
            builder.append(" ");
            builder.append(f.comparisonOperator);
            builder.append(" ");
            builder.append(f.value);
        }

        builder.append(")");

        return builder.toString();
    }
}
```

When Example 10.5 on page 210 is run it outputs these messages (annotations added):

```
#
# Unique query on managed object B
#
INFO: getSingleResult select obj from com.kabira.snippets.store.QueryNotifier$B using
ByName where (m_name == 22); Java constructor: (none)
INFO:    select object from B using ByName where (m_name == 22) with NOLOCK

#
# Maximum result query on managed object A
#
INFO: Get Maximum Result
INFO:    select maximum from A using ByName with READLOCK

#
# Minimum result query with a start range on managed object A
#
INFO: Get Minimum Result for obj in com.kabira.snippets.store.QueryNotifier$A using ByName
 where (m_name >= 8) { }
INFO:    select minimum from A using ByName where (m_name >= 8) with NOLOCK

#
# Range query on managed object B
#
INFO: Range Query for obj in com.kabira.snippets.store.QueryNotifier$B using ByName where
 (m_name > 22 && m_name <= 28) { }
INFO:    select object from B using ByName where (m_name > 22 && m_name <= 28) with NOLOCK
```

Atomic create or select

The ability to atomically either select or create a uniquely keyed object (see the section called "Atomic Create or Select" on page 82 for more details) is supported in the secondary store notifiers by a series of notifier calls depending on whether the object exists in shared memory after the `Query<T>.query` method is called. If the object exists, no other notifier entry points are called. If the object does not exist, the object is created in shared memory by the TIBCO ActiveSpaces® Transactions runtime and then the `Record<T>.created` notifier is called.

This behavior is shown in Example 10.6 on page 215.

Example 10.6. Atomic create or select

```
//     $Revision: 1.1.2.2 $
package com.kabira.snippets.store;

import com.kabira.platform.KeyFieldValueList;
import com.kabira.platform.KeyManager;
import com.kabira.platform.KeyQuery;
import com.kabira.platform.LockMode;
import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Key;
import com.kabira.platform.annotation.Managed;
import com.kabira.store.KeyField;
import com.kabira.store.Query;
import com.kabira.store.Record;
import java.util.ArrayList;

/**
 * Secondary store notifier behavior with an atomic select or create query
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class AtomicCreateSelect
{
    @Managed
    @Key(name = "ByName", unique = true, ordered = true, fields =
    {
        "m_name"
    })
    private static class A
    {
        private A(final String name)
        {
            m_name = name;
        }

        String getName()
        {
            return m_name;
        }

        private final String m_name;
    };

    private static class RecordNotifier extends Record<A>
    {
        public RecordNotifier()
        {
            super(A.class);
        }

        @Override
        public void created(A a)
        {
            System.out.println("INFO:\tCreated " + a.getName());
        }
    }

    private static class QueryNotifier extends Query<A>
    {
        public QueryNotifier()
        {
            super(A.class, "ByName");
        }
```

```
       @Override
      public void query(Class<? extends A> klass, LockMode lockMode, ArrayList<KeyField>
queryData)
       {
           assert queryData.size() == 1 : queryData.size();
           assert queryData.get(0).name.equals("m_name") : queryData.get(0);
           assert queryData.get(0).value instanceof String : queryData.get(0);
           String value = (String) queryData.get(0).value;

           if (m_create == true)
           {
               System.out.println("INFO:\tQuery creating object for " + value);
               new A(value);
               m_create = false;
           }
           else
           {
               System.out.println("INFO:\tQuery NOT creating object for " + value);

           }
       }

       @Override
      public void queryMinimum(Class<? extends A> klass, LockMode lm, ArrayList<KeyField>
al)
       {
           assert false : "queryMinimum";
       }

       @Override
      public void queryMaximum(Class<? extends A> klass, LockMode lm, ArrayList<KeyField>
al)
       {
           assert false : "queryMaximum";
       }

       private boolean m_create = true;
   }

   /**
    * Main entry point
    * @param args Not used
    */
   public static void main (final String [ ] args)
   {
       new Transaction("Initialize")
       {

           @Override
           protected void run()
           {
               new RecordNotifier();
               new QueryNotifier();
           }
       }.execute();

       new Transaction("Create or Select")
       {
           @Override
           protected void run()
           {
               KeyManager<A> km =  new KeyManager<>();
               KeyQuery<A> kq = km.createKeyQuery(A.class, "ByName");
               KeyFieldValueList values = new KeyFieldValueList();

               values.add("m_name", "foo");
```

```
                kq.defineQuery(values);

                System.out.println("INFO: " + kq);
                A a = kq.getOrCreateSingleResult(LockMode.WRITELOCK, null);
                System.out.println("INFO: Returned " + a.getName());

                values.clear();
                values.add("m_name", "bar");
                kq.defineQuery(values);

                System.out.println("INFO: " + kq);
                a = kq.getOrCreateSingleResult(LockMode.WRITELOCK, null);
                System.out.println("INFO: Returned " + a.getName());
            }
        }.execute();
    }
}
```

Example 10.6 on page 215 outputs these message when run (annotation added):

```
#
#  Query<T>.query creates object in shared memory
#
INFO: select obj from com.kabira.snippets.store.AtomicCreateSelect$A using ByName where
 (m_name == foo); Java constructor: (none)
INFO:    Query creating object for foo
INFO: Returned foo

#
#  Query<T>.query does not create object in shared memory
#  Object is created by the runtime and the Record<T>created
#  notifier is called
#
INFO: select obj from com.kabira.snippets.store.AtomicCreateSelect$A using ByName where
 (m_name == bar); Java constructor: (none)
INFO:    Query NOT creating object for bar
INFO: Returned bar
INFO:    Created bar
```

Chaining notifiers

Notifiers can be chained by a secondary store implementation. Notifier chaining is accomplished using these APIs:

- `Extent<T>.getNotifier(...)` - get currently installed `Extent<T>` notifier.
- `Query<T>.getNotifier(...)` - get currently installed `Query<T>` notifier for a specific key.
- `Record<T>.getNotifier(...)` - get currently installed `Record<T>` notifier.

Example 10.7 on page 217 shows the chaining of notifiers.

Example 10.7. Notifier chaining

```
//     $Revision: 1.1.2.2 $
package com.kabira.snippets.store;

import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;
import com.kabira.store.Record;

/**
 * Chaining secondary store notifiers
```

```
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class Chaining
{
    @Managed
    private static class A
    {
    };

    private static class Notifier extends Record<A>
    {
        private static void installNotifer()
        {
            //
            //    Get previously installed notifier (if any)
            //    This must be done before we create a new notifier
            //
            Record<A> previous = (Record<A>) Record.getNotifier(A.class);
            new Notifier(previous);
        }

        private Notifier(Record<A> previous)
        {
            super(A.class);

            //
            //    Save off previous notifier (may be null)
            //
            m_chained = previous;
        }

        @Override
        public void created(A a)
        {
            System.out.println("INFO: created " + a);

            //
            //    Call chained notifer (if any)
            //
            if (m_chained != null)
            {
                m_chained.created(a);
            }
        }
        private final Record<A> m_chained;
    }

    /**
     * Main entry point
     *
     * @param args Not used
     */
    public static void main(final String[] args)
    {
        new Transaction("Initialization")
        {
            @Override
            protected void run()
            {
                Notifier.installNotifer();
                Notifier.installNotifer();
                Notifier.installNotifer();
            }
```

```
        }.execute();

        new Transaction("Create Objects")
        {

            @Override
            protected void run() throws Transaction.Rollback
            {
                new A();
            }
        }.execute();
    }
}
```

Example 10.7 on page 217 outputs these messages when executed:

```
INFO: created com.kabira.snippets.store.Chaining$A@fce9ff01
INFO: created com.kabira.snippets.store.Chaining$A@fce9ff01
INFO: created com.kabira.snippets.store.Chaining$A@fce9ff01
```

11

Components

This chapter describes how to define and use TIBCO ActiveSpaces® Transactions components.

Defining a component

A component is a JAR file that contains a property file named `ast.properties`. The properties supported in the `ast.properties` file are summarized in Table 11.1 on page 221.

Table 11.1. Component properties

Property	Description
`Component-Name`	A unique indentifier for this component. This property is used both for identifying the component in log messages and to prevent the same component from being loaded multiple times. This property is required.
`Component-Notifiers`	A comma separated list of fully qualified class names which extend `com.kabira.platform.component.Notifier`. This property is optional.
`Configuration-Files`	A comma separated list of configuration files which include path information relative to the top of the JAR. It is recommended that configuration files are stored with enough path to prevent collisions with files of the same name in other components. This property is optional.

Property values must be specified in a single line unless the "\" line continuation character is used to span lines. See Example 11.1 on page 222 for an example property file.

White-space is allowed in the value for the `Component-Name` property.

All other white-space in property files is ignored.

Example 11.1. Component property file

```
#     $Revision: 1.1.2.1 $

#
#   The name of this component
#
Component-Name Snippet Component Sample

#
#   Notifiers associated with this component
#
Component-Notifiers com.kabira.snippets.components.NotifierOne, \
                        com.kabira.snippets.components.NotifierTwo

#
#   Configuration files associated with this component
#
Configuration-Files default.kcs
```

Locating the ast.properties file

The `ast.properties` file is searched for in these locations:

- top of the JAR file
- any element in the classpath

If the `ast.properties` file is located in the top of the JAR file it is always found and loaded if the JAR file is in the current class path. This is the simplest and most transparent mechanism to locate the `ast.properties` file.

It is also possible to locate the `ast.properties` file in a nested directory in a JAR file if this directory is specified in the class path.

If there are multiple `ast.properties` file located in a JAR file, the first one located is used. All others are ignored.

This is an example of a JAR file that contains the `ast.properties` file at the top of the JAR file.

Example 11.2. Location of ast.properties in JAR file

```
jar tf snippets.jar

META-INF/
META-INF/MANIFEST.MF
com/
com/kabira/
com/kabira/snippets/
com/kabira/snippets/components/
com/kabira/snippets/development/
com/kabira/snippets/distributedcomputing/
com/kabira/snippets/highavailability/
com/kabira/snippets/managedobjects/
com/kabira/snippets/management/
com/kabira/snippets/reference/
com/kabira/snippets/transactions/
com/kabira/snippets/vmlifecycle/

...

ast.properties
```

Notifiers

Component notifiers provide a mechanism to transparently perform component initialization and termination. The user of the component does not need to explicitly initialize or terminate a component.

A notifier is implemented by extending `com.kabira.platform.Notifier` and overriding the methods are that are needed to perform component initialization and termination. See Example 11.3 on page 223 for an example notifier.

Example 11.3. Component notifier

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.components;

import com.kabira.platform.component.Notifier;
import com.kabira.platform.component.ComponentException;

/**
 * Component notifier one
 */
public class NotifierOne extends Notifier
{
    @Override
    protected void preConfigurationInitialize()
        throws ComponentException
    {
        try
        {
            //
            //  Resolve configuration class
            //
            Class.forName("com.kabira.snippets.components.Defaults");
        }
        catch (ClassNotFoundException ex)
        {
            new ComponentException("Class resolution failed", ex);
        }

        m_notifierName = getClass().getSimpleName();

        System.out.println(m_notifierName + ": - preConfigurationInitialize");
    }

    @Override
    protected void postConfigurationInitialize()
    {
        System.out.println(m_notifierName + ": - postConfigurationInitialize");

    }

    @Override
    protected void preConfigurationTerminate()
    {
        System.out.println(m_notifierName + ": - preConfigurationTerminate");

    }

    @Override
    protected void postConfigurationTerminate()
    {
        System.out.println(m_notifierName + ": - postConfigurationTerminate");
    }
```

```
    private String  m_notifierName;
}
```

Notifiers are created and executed in the order they are specified in the `Component-Notifiers` property during component initialization. They are executed and released for garbage collection in the reverse order during component termination. Since the same notifier instance is used for initialization and termination, state information can be stored in the notifier during the lifetime of the component for implementation specific reasons.

Component activation fails if:

- a notifier class specified in the `Component-Notifiers` property cannot be found.
- a notifier class does not extend from `com.kabira.platform.component.Notifier`.
- a `com.kabira.platform.component.ComponentException` is thrown by a notifier initialization method.

Configuration

Components may contain configuration files. The configuration files contained in a component are automatically loaded and activated when a component is initialized. They are loaded and activated in the order they are specified in the `Configuration-Files` property. The configuration files are deactivated and removed when the component is terminated. They are deactivated and removed in the reverse order from which they were activated during initialization.

Component activation fails if:

- a configuration file specified in the `Configuration-Files` property cannot be found.
- a configuration file load fails.
- a configuration file activation fails.

Any failures deactivating and removing configuration files during component termination are ignored.

Example

This is a complete example of a component. The example consists of these files:

- `NotifierOne.java` - component notifier one (see Example 11.3 on page 223).
- `NotifierTwo.java` - component notifier two (see Example 11.4 on page 224).
- `Defaults.java` - example configuration definition (see Example 11.5 on page 225).
- `default.kcs` - configuration file loaded by component (see Example 11.6 on page 225).
- `ComponentMain.java` - driver to execute example (see Example 11.7 on page 226).
- `ast.properties` - component property file (see Example 11.1 on page 222).

Example 11.4. NotifierTwo.java

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.components;
```

```
import com.kabira.platform.component.Notifier;

/**
 * Component notifier two
 */
public class NotifierTwo extends Notifier
{
    @Override
    protected void preConfigurationInitialize()
    {
        m_notifierName = getClass().getSimpleName();

        System.out.println(m_notifierName + ": - preConfigurationInitialize");
    }

    @Override
    protected void postConfigurationInitialize()
    {
        System.out.println(m_notifierName + ": - postConfigurationInitialize");

    }

    @Override
    protected void preConfigurationTerminate()
    {
        System.out.println(m_notifierName + ": - preConfigurationTerminate");

    }

    @Override
    protected void postConfigurationTerminate()
    {
        System.out.println(m_notifierName + ": - postConfigurationTerminate");
    }

    private String  m_notifierName;
}
```

Example 11.5. Defaults.java

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.components;

import com.kabira.platform.kcs.Configuration;

/**
 * Default component configuration
 */
public class Defaults extends Configuration
{
    String  defaultValue;
}
```

Example 11.6. default.kcs

```
//     $Revision: 1.1.2.1 $
configuration "default" version "1.0" type "com.kabira.snippets.components"
{
        configure com.kabira.snippets.components
        {
                Defaults
                {
                        defaultValue = "hello world";
                };
```

```
        };
};
```

Example 11.7. ComponentMain.java

```
//     $Revision: 1.1.2.1 $
package com.kabira.snippets.components;

import com.kabira.platform.Transaction;
import com.kabira.platform.ManagedObject;


/**
 * Execute component example
 *
 * NOTE: This example requires the classpath to explicitly include the
 * directory that contains this snippet source so that the ast.properties
 * file is found.
 *
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class ComponentMain
{
    /**
     * @param args  None supported
     */
    public static void main(String[] args)
    {
        System.out.println("Main executed");

        new Transaction("Components")
        {
            @Override
            protected void run() throws Rollback
            {
                for (Defaults defaults : ManagedObject.extent(Defaults.class ))
                {
                    System.out.println(
                        "Configured default value: " + defaults.defaultValue);
                }
            }

        }.execute();

        System.out.println("Main exiting");
    }
}
```

When `main` is executed the following output (annotation added) is generated.

This example requires that snippet be run using a JAR file containing the `ast.properties` file.

Example 11.8. Example component output

```
#
#
#   Component initialization
#
[A] NotifierOne: - preConfigurationInitialize
```

```
[A] NotifierTwo: - preConfigurationInitialize
[A] NotifierOne: - postConfigurationInitialize
[A] NotifierTwo: - postConfigurationInitialize

#
#   Main execution
#
[A] Main executed
[A] Configured default value: hello world
[A] Main exiting

#
#   Component termination
#
[A] NotifierTwo: - preConfigurationTerminate
[A] NotifierOne: - preConfigurationTerminate
[A] NotifierTwo: - postConfigurationTerminate
[A] NotifierOne: - postConfigurationTerminate
```

12

System management

This chapter describes how to define system management targets.

Defining a system management target

A system management target is defined by extending `com.kabira.platform.management.Target`.

Example 12.1. Defining a management target

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.management;

import com.kabira.platform.management.Target;
import com.kabira.platform.management.ManagementTarget;

/**
 * MyTarget management target definition
 */
@ManagementTarget(name = "mytarget", description = "a management target")
public class MyTarget extends Target
{
}
```

Annotations are used to define the user interface to the management target. The annotations used are:

Table 12.1. System management annotations

Annotation	Description
@ManagementTarget	Define the name and description of a management target.
@Command	Indicate that a method should be exposed as a management command.
@Parameter	Define a management command parameter.

@Default	Define default values for management command parameters.

@ManagementTarget annotation

The `@ManagementTarget` annotation, and it usage, is defined as:

Example 12.2. @ManagementTarget annotation

```
package com.kabira.platform.management;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * Indicate that this class implements a Management Target.  This
 * annotation must be applied to any class registered with Target.register().
 */
@Documented
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface ManagementTarget
{
     /**
     * The public name of the target.  Required.
     */
    String name();

    /**
     * The description of the target. Used to generate command help message.
     */
    String description() default "";
}

//
//  Using the @ManagementTarget annotation
//
@ManagementTarget(name = "mytarget", description = "a management target")
public class MyTarget extends Target
{
}
```

@Command annotation

The `@Command` annotation, and its usage, is defined as:

Example 12.3. @Command annotation

```
package com.kabira.platform.management;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
```

```
 * Indicate that this method should be exposed as an command.
 * Commands must return void, or TargetException will be thrown
 * when the target is registered.
 */
@Documented
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Command
{
    /**
     * The description of the command. Used to generate command help message.
     */
    String description() default "";
}

//
//  Using the @Command annotation
//
public class MyTarget extends Target
{
    @Command(description = "a management command")
    public void setstate() { ... }
}
```

A method exposed as a management command must be:

- `public`
- return `void`

A `TargetException` is thrown when the management target is registered if a method with the `@Command` annotation does not following these rules.

@Default annotation

The `@Default` annotation, and its usage, is defined as:

Example 12.4. @Default annotation

```
package com.kabira.platform.management;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * Defines default values for parameters.
 * see java.lang.String for descriptions of valid String formats for
 * the supported parameter types.
 */
@Documented
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Default
{
    /**
     * Must be set true if the default value is provided.
     */
```

```
    boolean provided() default true;

    /**
     * The default value to be used.
     */
    String value();
}

//
//  Using the @Default annotation
//
public class MyTarget extends Target
{
    public void setstate(
       @Parameter(name = "state", defaultValue = @Default(value = "true")) String state)
 { ... }
}
```

@Parameter annotation

The `@Parameter` annotation, and its usage, is defined as:

Example 12.5. @Parameter annotation

```
package com.kabira.platform.management;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * ManagementTarget Command parameter.
 * Must be present on all parameters of a method annotated as a @Command. Valid
 * parameter types are String, Enum, Boolean, Character, Integer, Byte, Short,
 * Long, Float, and Double.  Other types will cause TargetException to be
 * thrown when the target is registered.  Arrays are not supported.
 * see java.lang.String for descriptions of valid String formats for
 * the supported parameter types.
 */
@Documented
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.PARAMETER)
public @interface Parameter
{
    /**
     * The parameter name as exposed to management clients
     */
    String name();

    /**
     * A description of the parameter. Used in the generated help for the
     * command.
     */
    String description() default "";

    /**
     * Indicates that this parameter must be set by the caller.
     * If the parameter is not set. the command will fail.
     * If false, and no default is provided, the value of the parameter will be
     * null
     */
```

```
    boolean required() default false;

    /**
     * The default value of the parameter.
     */
    Default defaultValue() default @Default(provided = false, value = "");
}

//
//  Using the @Parameter annotation
//
public class MyTarget extends Target
{
    public void setstate(
        @Parameter(name = "state",
            description="state value",
            required=true,
            defaultValue = @Default(value = "true")) String state) { ... }
}
```

All parameters of a method marked with the `@Command` annotation must be marked with the `@Parameter` annotation. The supported parameter types are:

- `String`
- `Enum`
- `Boolean`
- `Character`
- `Integer`
- `Byte`
- `Short`
- `Long`
- `Float`
- `Double`

A `TargetException` is thrown when the management target is registered if an unsupported parameter type is detected.

Synchronous and asynchronous execution

Administrative commands can execute synchronously or asynchronously. In both cases, the command appears synchronous from the caller's perspective. The difference is whether the command implementation completes immediately or requires multiple transactions to complete. Synchronous commands are appropriate for commands that can be immediately executed and do not lock a large number of resources, for example a display command. Asynchronous commands are appropriate for commands that require a large amount of processing, or would lock a large number of objects into a single transaction, for example a bulk load of a file into memory.

A synchronous command is one which calls `Target.commandComplete()` or `Target.commandFailed()` before returning from the command implementation.

An asynchronous command returns without calling `Target.commandComplete()` or `Target.commandFailed()` from the command implementation. In this case, the command is considered active until one of these methods is called sometime later in another transaction. It is important that one of these methods be called at some later time, or this command will never complete.

Asynchronous commands must call `Target.commandComplete()` or `Target.commandFailed()` on the target instance on which the asynchronous command was initiated.

Security

Administrative commands are only executed following a successful authentication. The principal that is executing a command can be determined using `Target.getActivePrincipalName()`.

Access control should be configured for all application administration targets. This protects the application from unauthorized access to administrative functions. In general, access to any command that modifies the state of the running system should only be granted to the `switchadmin` role. Commands that only display system state should grant access to the `switchmonitor` role. See the **TIBCO ActiveSpaces® Transactions Administration** for details on the `switchadmin` and `switchmonitor` roles.

An example security configuration for an administration target is in Example 12.10 on page 239.

Conventions

The following conventions are recommended for management targets:

- target names are nouns.
- command names are verbs.
- target, command, and parameter names are all lower case.
- single word target, command, and parameter names are preferred.
- there are no spaces, or underscores, used in multi-word target, command, or parameter names.
- most targets define a `display` command which returns tabular data describing the target's state.

Target lifecycle

A management target must be registered before it is available for use. It should be unregistered when the target is no longer needed. Generally all management targets should be registered when an application starts and be unregistered when the application exits.

A new target object instance is created by the management framework for each command invocation. The instance is created in the JVM in which the management target was registered. This means that the command will execute in this JVM.

Since a new target object instance is created for each command, asynchronous commands can store command state in fields in the target object. The target object instance is destroyed by the management framework when the command completes.

Example

This is a complete example of a management target implementation. The example consists of these files:

- `AnEnum.java` - definition of an enumeration used as a parameter.
- `ExampleTarget.java` - target implementation.
- `ExampleMain.java` - driver to execute example target.
- `ExampleTargetLifecycle.java` - initialize and terminate the example target.
- `exampletargetsecurity.kcs` - security definition for example target.

Example 12.6. AnEnum.java

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.management;

/**
 * A simple enum for the example target
 */
public enum AnEnum
{
    /**
     * for the money
     */
    One,
    /**
     * for the show
     */
    Two,
    /**
     * to get ready
     */
    Three,
    /**
     * to go!
     */
    Four
};
```

Example 12.7. ExampleTarget.java

```
//     $Revision: 1.1.2.2 $
package com.kabira.snippets.management;

import com.kabira.platform.annotation.Asynchronous;
import com.kabira.platform.management.Command;
import com.kabira.platform.management.Default;
import com.kabira.platform.management.ManagementTarget;
import com.kabira.platform.management.Parameter;
import com.kabira.platform.management.Target;
import com.kabira.platform.management.TargetError;

/**
 * An example of an management target
 */
@ManagementTarget(
    name = "exampletarget",
    description = "example management target")
```

```
public class ExampleTarget extends Target
{
    /**
     * example display command. return a simple table of data.
     *
     * @throws TargetError
     */
    @Command
    public void display() throws TargetError
    {

        String[] names =
        {
            "Who", "Where", "What"
        };
        setColumnNames(names);

        String[] row =
        {
            "Colonel Mustard", "Library", "Candlestick"
        };
        addRow(row);

        commandComplete();
    }

    /**
     * example of command which fails. Transactions are committed when
     * commandFailed is called.
     */
    @Command
    public void setstate()
    {
        commandFailed("setstate not supported");
    }

    /**
     *
     * example of command which throws an exception. If an exception is thrown,
     * the command fails, and the transaction is aborted.
     *
     * @throws Exception
     */
    @Command
    public void rollbackexample() throws Exception
    {
        throw new Exception("transaction will roll back");
    }

    /**
     * Example of Parameter annotations
     *
     * @param version required String parameter
     * @param count optional Integer parameter with default value
     * @param enum123 optional Enum parameter with default value
     * @param aBool optional Boolean parameter with default value
     * @param aFloat optional Float parameter with default value
     */
    @Command
    public void examplecommand(
        @Parameter(name = "version", description = "version number",
            required = true)
        String version,
        @Parameter(name = "count", description = "number to display",
            defaultValue = @Default(value = "1"))
        Integer count,
        @Parameter(name = "enum123", description = "Pick One Two or Three",
```

```
            defaultValue = @Default(value = "One"))
        AnEnum enum123,
        @Parameter(name = "abool", defaultValue = @Default(value = "true"))
        Boolean aBool,
        @Parameter(name = "afloat", defaultValue = @Default(value = "123.456"))
        Float aFloat)
    {

        commandComplete();
    }

    /**
     * Asynchronous command example. asyncCommand() starts the command.
     *
     * @throws TargetError
     */
    @Command
    public void asynccommand() throws TargetError
    {
        m_asyncStartTime = System.currentTimeMillis();
        asyncCommandComplete();
    }

    /**
     * Complete the asynchronous command.
     */
    @Asynchronous
    private void asyncCommandComplete() throws TargetError
    {
        long elapsedTime = System.currentTimeMillis() - m_asyncStartTime;
        String[] names =
        {
            "Elapsed Time (Milli-seconds)"
        };
        setColumnNames(names);
        String[] row =
        {
            Long.toString(elapsedTime)
        };
        addRow(row);
        commandComplete();
    }
    private long m_asyncStartTime;
}
```

Example 12.8. ExampleMain.java

```
//     $Revision: 1.1.2.2 $
package com.kabira.snippets.management;

import com.kabira.test.management.Client;
import com.kabira.test.management.CommandFailed;
import java.util.HashMap;

/**
 * Execute example target
 * <p>
 * <h2> Target Nodes</h2>
 * <ul> <li> <b>domainnode</b>= A </ul>
 */
public class ExampleMain
{
    /**
     * @param args None supported
     * @throws CommandFailed Command execution failed
     */
```

```
    public static void main(String[] args) throws CommandFailed
    {
        //
        //  Initialize target
        //
        ExampleTargetLifecycle.register();

        //
        //  Create test client
        //
        Client client = new Client("guest", "guest");

        //
        //  Execute help on the target
        //
       String[] results = client.runCommand("help", ExampleTargetLifecycle.Name, null);


        //
        //  Display results
        //
        for (String s : results)
        {
            System.out.println(s);
        }

        //
        //  Execute display
        //
        results = client.runCommand("display", ExampleTargetLifecycle.Name, null);

        //
        //  Display results
        //
        for (String s : results)
        {
            System.out.println(s);
        }

        //
        //  Execute the asynchronous command
        //
        results = client.runCommand("asynccommand", ExampleTargetLifecycle.Name, null);

        //
        //  Display results
        //
        for (String s : results)
        {
            System.out.println(s);
        }

        //
        //  Execute examplecommand
        //
        HashMap<String, String> parameters = new HashMap<String, String>();
        parameters.put("version", "1.0");
        parameters.put("count", "100");
        parameters.put("enum123", "Three");
        parameters.put("abool", "true");
        parameters.put("afloat", "3.75");
        client.runCommand("examplecommand", ExampleTargetLifecycle.Name, parameters);

        //
        //  Terminate target
        //
        ExampleTargetLifecycle.unregister();
```

```
    }
}
```

Example 12.9. ExampleTargetLifecycle.java

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.management;

import com.kabira.platform.management.Target;
import com.kabira.platform.Transaction;

/**
 * Example target life-cycle implementation
 */
public class ExampleTargetLifecycle
{
    /**
     * Target name
     */
    public final static String Name = "exampletarget";

    /**
     * Register MyTarget management target
     */
    public static void register()
    {
        new Transaction("Register Target")
        {

            @Override
            protected void run() throws Rollback
            {
                Target.register(ExampleTarget.class);
            }
        }.execute();

    }

    /**
     * Un-register MyTarget management target
     */
    public static void unregister()
    {
        new Transaction("Un-register Target")
        {

            @Override
            protected void run() throws Rollback
            {
                Target.unregister(Name);
            }
        }.execute();
    }
}
```

Example 12.10. exampletargetsecurity.kcs

```
//    $Revision: 1.1.2.1 $

configuration "exampletargetsecurity" version "1" type "security"
{
 configure security
 {
  configure AccessControl
  {
```

```
   //
   // this rule locks all elements in
   // ExampleTargetCommand to prevent unauthenticated
   // access, and then grants full access to all
   // elements in the command to the "switchadmin" role.
   //
   Rule
   {
    name = "com.kabira.platform.management.ExampleTarget";
    lockAllElements = true;
    accessRules =
    {
     {
      roleName = "switchadmin";
      permission = AccessAllOperationsAndAttributes;
     }
    };
   };

   //
   // the remaining rules grant access to specific
   // operations to the "switchmonitor" role:
   //
   Rule
   {
    name = "com.kabira.platform.management.ExampleTarget.examplecommand";
    accessRules =
    {
     {
      roleName = "switchmonitor";
      permission = Execute;
     }
    };
   };
  };
 };
};
```

When main is executed the following is output (annotation added):

Example 12.11. Example Target Output

```
#
#     Output from help command
#
[A]   valid commands and parameters for target "exampletarget":
[A]
[A]   display exampletarget
[A]
[A]   asynccommand exampletarget
[A]
[A]   examplecommand exampletarget
[A]     version=<String>
[A]         version number
[A]
[A]     [ count=<Integer, default = 1> ]
[A]         number to display
[A]
[A]     [ enum123=<AnEnum, default = One> ]
[A]         Pick One Two or Three
[A]
[A]     [ abool=<Boolean, default = true> ]
[A]
[A]     [ afloat=<Float, default = 123.456> ]
[A]
[A]
```

```
[A]   setstate exampletarget
[A]
[A]   rollbackexample exampletarget
[A]
[A]
[A] Description:
[A]
[A] example management target

#
#     Output from display command
#
[A] Who    Where    What
[A] Colonel Mustard    Library    Candlestick

#
#     Output from asynccommand command
#
[A] Elapsed Time (Milli-seconds)
[A] 3
```

13

Monitoring applications

This chapter briefly describes how to monitor applications. See the **TIBCO ActiveSpaces® Transactions Administration** for complete details on administration.

Management console

The management console - TIBCO ActiveSpaces® Transactions Administrator - is used to control and monitor TIBCO ActiveSpaces® Transactions nodes. The following steps must be taken to start monitoring a management domain.

1. Connect to the TIBCO ActiveSpaces® Transactions Administrator URL with a Web Browser.

2. Log into the management console using a username of `guest` and a password of `guest`.

The sections below show these steps in more detail.

Management console login

The initial screen displayed when navigating to the TIBCO ActiveSpaces® Transactions Administrator URL is:

Figure 13.1. Manager login screen

Following a successful login and selecting an application node, this screen is displayed:

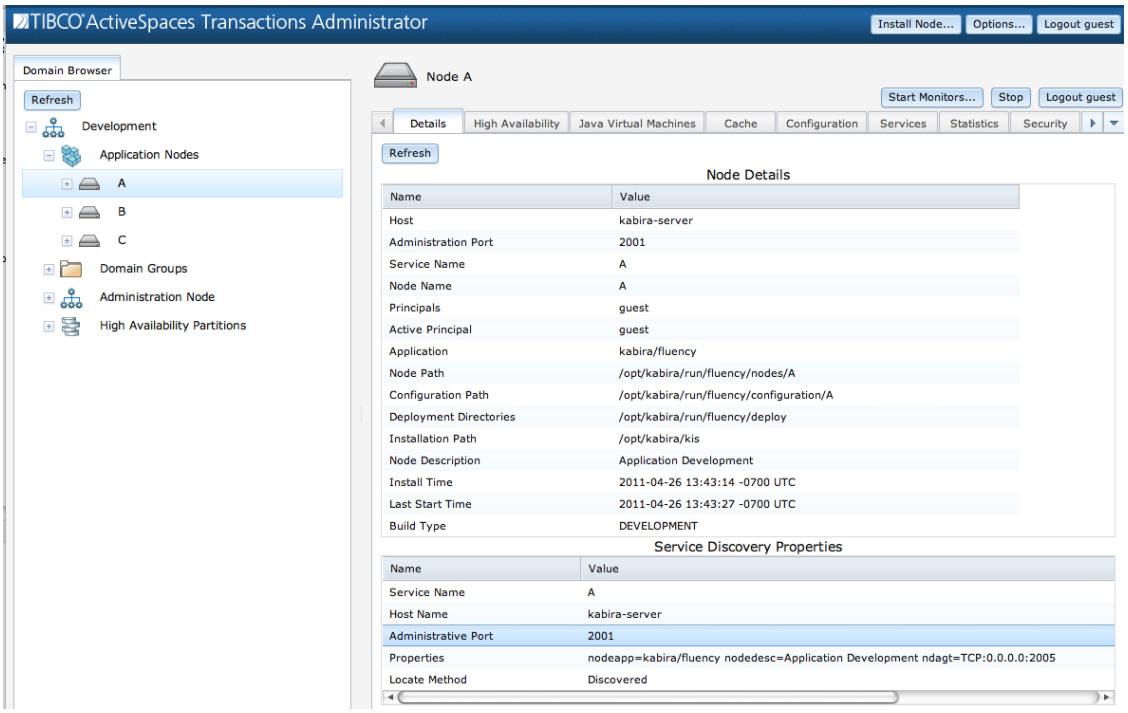

Figure 13.2. Node details display

Object monitor

The Object Monitor allows viewing of Managed Objects in shared memory. The Object Monitor is started from the command line using the `monitor` executable.

Figure 13.3 shows the object monitor display. An instance of the `com.kabira.snippets.highavailabilty.FlintStone` class is being displayed. The value of the name field is displayed as well as the partition in which the object is currently assigned, partition `Odd` in this case.

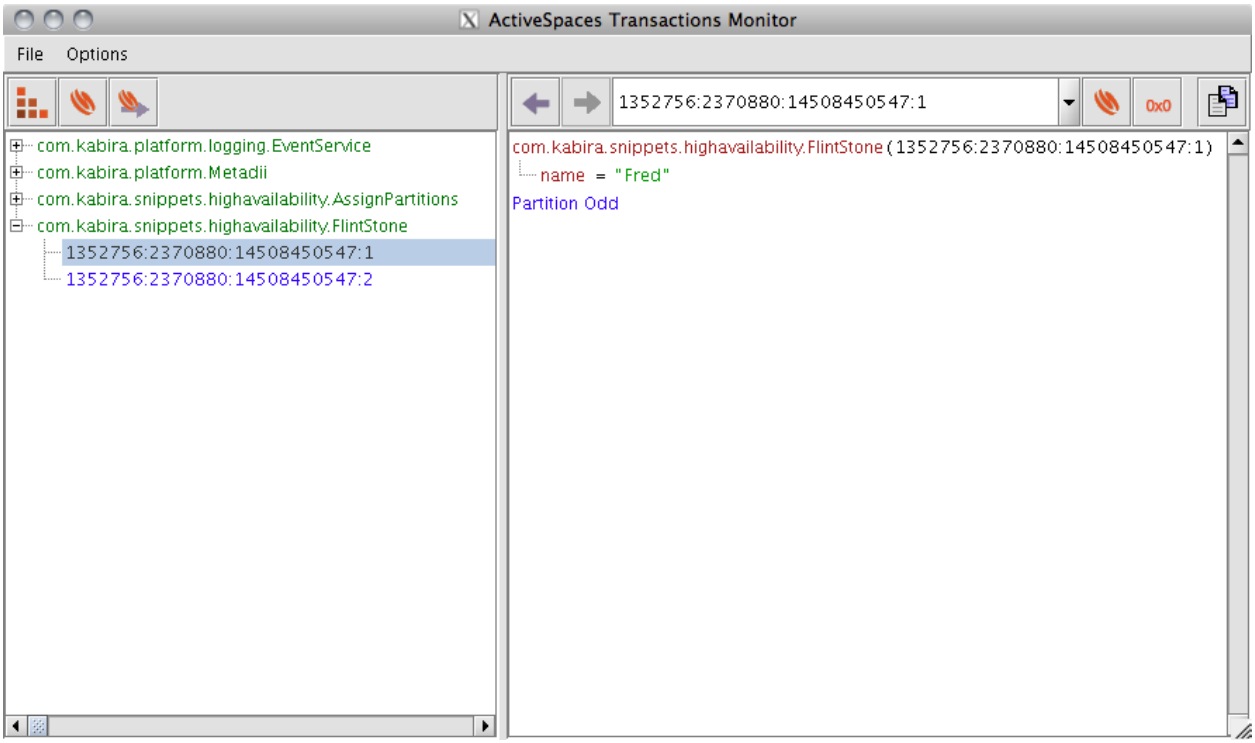

Figure 13.3. Object monitor

Clicking on the partition name associated with an object brings up the information on the partition in Figure 13.4. This screen displays this information about the partition:

- **Name** - the partition name is in the `m_name` field.
- **Node List** - the current node list for the partition is in the `m_nodeNames` array. The first node in the list is the active node for the partition.
- **State** - the current partition state is in the `m_state` field.
- **Last State Change Time** - the last time the partition state changed is in the `m_lastUpdated` field.

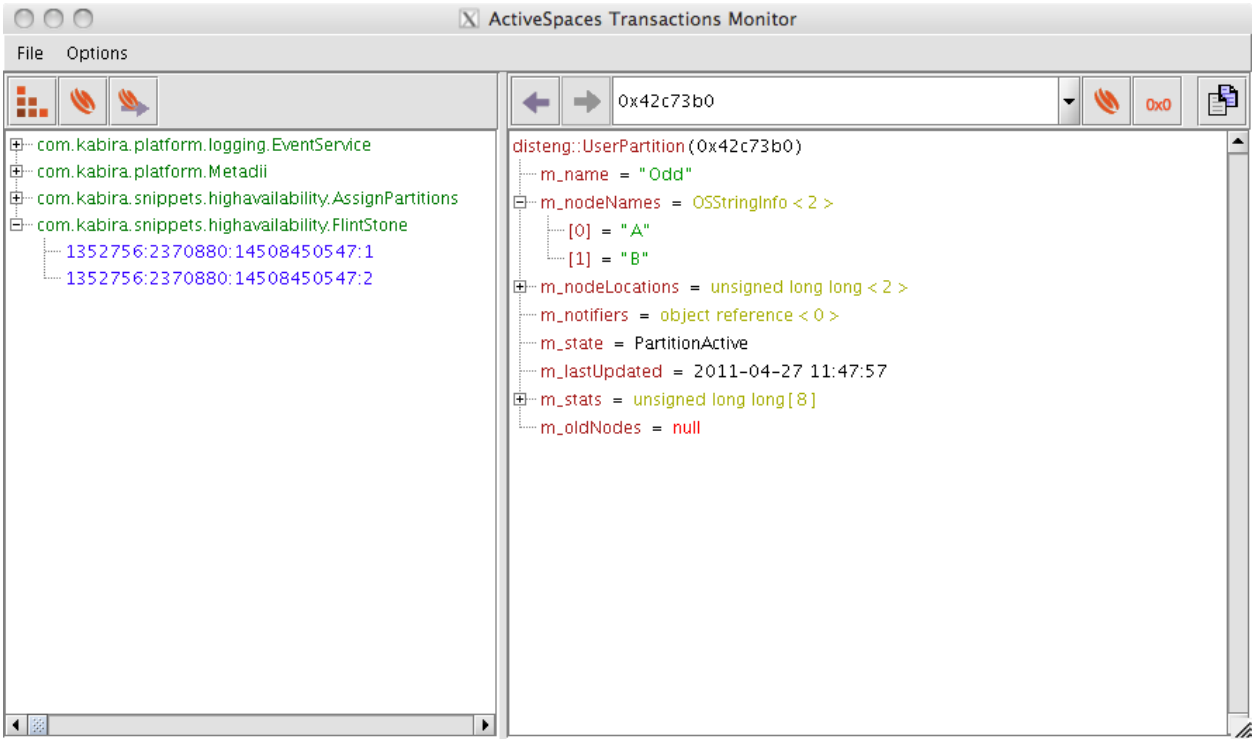

Figure 13.4. Object partition details

14

Reference

This chapter contains reference material for TIBCO ActiveSpaces® Transactions .

Class resolution

TIBCO ActiveSpaces® Transactions uses these locations to resolve a class reference. The locations are searched in the following order:

1. TIBCO ActiveSpaces® Transactions defined system `CLASSPATH`
2. JAR files located in deployment directories
3. Client side `CLASSPATH` definition, if `detach=false` (see Table 14.1 on page 249) when an application is deployed.

The system `CLASSPATH` cannot be changed.

The contents of the deployment directories are then searched to resolve class resolutions as described in **TIBCO ActiveSpaces® Transactions Administration**.

Finally the client's `CLASSPATH` is searched. All classes resolved using the client class path are sent over a network connection to the server. The client `CLASSPATH` is only available for class resolution if the `detach` option is set to `false` when the application was deployed.

Client and server class paths are distinct. The client class path is never sent to the server. All resources in a client's class path that are in a shared file system visible to the server are still sent to the server from the client if the resource is not also in the server's class path.

Native libraries

Native libraries can be installed on an TIBCO ActiveSpaces® Transactions node using one of these techniques:

- using the `java.library.path` JVM property specified on the deployment tool (see the section called “Deployment tool” on page 248) command line.
- deployment directories. See the **TIBCO ActiveSpaces® Transactions Administration** for details on specifying deployment directories.

The directories specified using either of these methods are automatically added to the `LD_LIBRARY_PATH` of all JVMs started on the node. The directories are added to the `LD_LIBRARY_PATH` in this order:

1. The `java.library.path` directories specified on the deployment tool command-line.
2. The node's deployment directories.

The directories specified in the `java.library.path` property are server side directories. The directory names can be absolute or relative, and they are separated with a `:`. All non-absolute paths are relative to the node directory.

```
java -jar deploy.jar [-Djava.library.path=<dir1>[:...:<dirN>]] \
    <target> [application parameters]
```

Deployment tool

TIBCO ActiveSpaces® Transactions ships with a deployment tool that is used to deploy applications to TIBCO ActiveSpaces® Transactions nodes. The deployment tool can be used from the command line or via a Java IDE. The TIBCO ActiveSpaces® Transactions deployment tool is named `deploy.jar`.

The general syntax for using the deployment tool is:

```
java [local JVM options] -jar deploy.jar [options] <target> [application parameters]
java [local JVM options] -jar deploy.jar [options] help
```

`deploy.jar` must be specified as the first `-jar` option. This ensures that the deployment tool gets control during the execution of the application and manages execution and debugging of the application on an TIBCO ActiveSpaces® Transactions node.

Attempting to execute an application that uses TIBCO ActiveSpaces® Transactions classes without specifying the `deploy.jar` file as the first `-jar` option will cause a Java stack dump (example below) because the TIBCO ActiveSpaces® Transactions classes cannot execute outside of an TIBCO ActiveSpaces® Transactions JVM:

```
Exception in thread "main" java.lang.SecurityException: Prohibited package name:
 java.lang
at java.lang.ClassLoader.preDefineClass(ClassLoader.java:479)
at java.lang.ClassLoader.defineClass(ClassLoader.java:614)
at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:124)
at java.net.URLClassLoader.defineClass(URLClassLoader.java:260)
at java.net.URLClassLoader.access$100(URLClassLoader.java:56)
at java.net.URLClassLoader$1.run(URLClassLoader.java:195)
at java.security.AccessController.doPrivileged(Native Method)
at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
```

```
at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
at sun.misc.Launcher.loadClass(Launcher.java:268)
at java.lang.ClassLoader.loadClass(ClassLoader.java:251)
at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:319)
at pojopersistent.Main.main(Main.java:23)
```

[local JVM options] is used to pass options to the local JVM. These options are specific to the version of the local JVM.

[options] consists of any combination of JVM options (See the section called "Supported JVM properties" on page 256) or deployment tool options (See Table 14.1 on page 249). JVM options are prefixed with a "-", while deployment tool options are of the form *name* = *value*.

<target> is the application archive file, e.g. a JAR file, or class that will be executed on the node.

[application parameters] are application specific parameters that are passed to the application program's main.

The `help` command displays the usage message.

Table 14.1 on page 249 summarizes the supported deployment tool options:

Table 14.1. Deployment tool options

Option	Description
`adminport`	The administration port of the node that should be used to run the application.
`buildtype`	This option specifies the type of binaries used by the JVM. The valid values are `PRODUCTION` or `DEVELOPMENT`. If not specified, the JVM will use the same binary type as the node it runs in.
`copytarget`	A boolean flag which controls whether or not to copy the *<target>* parameter to the node when `detach=true`. If `copytarget=false`, *<target>* must exist in a deployment directory on the server. It is an error to set `copytarget=true` when `detach=false` or *<target>* is not an archive file, e.g. a JAR file. (default: `false`).
`debug`	An enumeration indicating the level of diagnostic output. The valid values are `enable` - a high level description of the deployment tool activity, `verbose` - detailed description of deployment tool activity, or `disable` - no diagnostic output. (default: `disable`).
`detach`	A boolean flag controlling the deployment mode. A value of `false` causes the deployment tool to operate in attached mode. A value of `true` causes the deployment tool to operate in detached mode. (default : `false`).
`detachtimeout`	The number of seconds to wait before exiting, when deploying in detached mode (`detach = true`). The state of the deployed application is validated for the specified timeout value before exiting (default: `10`).
`discoverytimeout`	The number of seconds to wait while resolving a `servicename` (default: `10`).
`displayversion`	A boolean flag indicating whether the product version information should be displayed (default: `true`).
`domainname`	The name of the domain that the application is to run on. When this option is used, the deployment tool must connect to a Domain Manager node

	which is managing the given domain. The application will execute on all nodes in the domain.
domaingroup	The name of the domain group that the application is to run on. When this option is used, the deployment tool must connect to a Domain Manager node which is managing the given domain group. The application will execute on all nodes in the domain group.
domainnode	The name of the domain node that the application is to run on. When this option is used, the deployment tool must connect to a Domain Manager node which is managing the given domain node. The application will execute on the specified node.
exitonfailure	Control multi-node completion behavior. A value of `true` causes the deployment tool to return to the caller when any node returns a non-zero exit code. A value of `false` causes the deployment tool to not return until all nodes exit. (default: `false`).
hostname	The host name running the node that should be used to run the application (default: `localhost`).
ignore-optionsfile	A boolean flag which can inhibit the default reading of the deploy tool options file (default: `false`).
jvmname	The administrative name to assign to the JVM which hosts the application being deployed. Deployment of the application fails if a JVM already exists with this name (default: a generated unique name). This name will also be used to search for JVM-specific deployment directories when the JVM starts. See **TIBCO ActiveSpaces® Transactions Administration** for information on JVM-specific deployment directories.
keepaliveseconds	The number of seconds that the node should wait before terminating a deployed application when connectivity between the node and the deploy tool is lost (default: 10 seconds).
mirrorclient	A boolean option, when given a value of `true`, insures that the working directory and class paths used by the node match the client. Note this option will only work if the client and node are run on the same machine, or reference identical file systems (default: `false`).
nodecleanup	A boolean flag that controls whether server artifacts (e.g. loaded class files) to start a JVM are removed when the JVM exits. A value of `true` removes all generated artifacts, a value of `false` does not. (default: `true`).
password	The password to use when authenticating `username` during the connection to the node.
remotedebug	If `true`, require the JVM hosting the application to enable remote debugging (default: `false`).
remotedebugport	The debugger agent port, to be used by the JVM to listen for remote debugger clients (default: randomly chosen by the JVM).
reset	This option, when given a value of `true`, requests that all Java objects and any changed type definitions on the node be deleted before the application begins execution. If there is a JVM already running an error will be reported and the new JVM will fail to start. (default: `false`).
schedulerpolicy	The Scheduling policy and priority for the JVM process. The syntax is *`policy`*`:` *`priority`*. The valid values for the policy are `SCHED_FIFO`, `SCHED_RR`, and `SCHED_OTHER`. The valid range for priority depends on

	the policy; for Linux the valid values for `SCHED_FIFO` and `SCHED_RR` are 1 - 99. See the man page page for `sched_setscheduler(2)` for more information.
	
	If the schedulerPolicy cannot be set, the JVM process will not be started. In order to use "real-time" policies, the caller needs the approriate permissions. On Linux, a security configuration file can be created to allow users to enable "real-time" policies. As an example, the following:
	
	`cat > /etc/security/limits.d/tibco.conf <<EOF`
	`nightly hard rtprio 70`
	`nightly soft rtprio 0`
	`EOF`
	
	will allow the user "nightly" to enable either SCHED_FIFO or SCHED_RR with a priority range from 0 to 70.
	
	If running as root, you can set the required permission for root, with the following command:
	
	`ulimit -r 99`
	
	Run this immediately before starting the node, in the same shell that starts the node.

`serverdebug`	Control server diagnostics. An enumeration value that has one of the following values: `enable` or `disable`. A value of `enable` causes additional server debug tracing. A value of `disable` disables server debug tracing. (default: `disable`).
`serverdebugfilter`	Internal. Control server diagnostics. A hexadecimal value, starting with 0x.
`serverdebugpause`	Internal. Pause process before JVM is launched. Boolean.
`servicename`	The service name of the node that is to be used to run the application. This option may be used instead of `adminport` and `hostname`.
`shutdowntimeoutseconds`	The maximum number of seconds to wait for a JVM to shutdown (default: 60).
`suspend`	If `true`, require the JVM to suspend execution before main() is called during remote debugging. This option only applies if `remotedebug=true` is specified (default: `false`).
`username`	The user name to use when connecting to a node. The specified value must identify a principal with administrative privileges on the node. Defaults to the `user.name` system property, which is set to be the UID of the user who executed the deployment tool client JVM.

Reset

The `reset` option provides development support for changing the shape of Managed Objects in shared memory. It has no affect on non-managed Java objects.

The `reset` option should only be used during development. It should never be used on a production system.

Reset processing takes place during JVM initialization. The following steps are taken to reset shared memory:

1. Check to make sure that there are no other JVMs running. If there are other JVMs running, the new JVM initialization will fail with an error message. Reset can only be done when there are no JVMs running on a node.

2. Delete all managed objects in shared memory. If the managed object is a distributed object on a remote node, dispatch the delete to the remote node. If the remote node cannot be reached, delete the cached copy on the local node.

3. Remove all type definitions from shared memory. They will be redefined as part of class loading as the application executes.

Examples of changing the shape of a Managed Object are:

- adding a field to a class
- removing a field from a class
- changing the type of a field in a class
- changing the inheritance hierarchy

TIBCO ActiveSpaces® Transactions detects when the shape of a Managed Object changes and fails the load of the changed class definition if `reset = false`.

For example, if Example 14.1 on page 252 is run twice - once with `m_string` not commented out, and then again with `m_string` commented out with `reset = false` (the default value):

Example 14.1. Type change

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.reference;

import com.kabira.platform.Transaction;
import com.kabira.platform.annotation.Managed;

/**
 * Snippet used to demonstrate changing the shape of a class
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class Shape
{
    /**
     * A managed object
     */
    @Managed
    public static class ShapeChange
    {
        private    int    m_int;
        //
        //    Uncomment this field after running once to see
        //    type conflict
        //
        // private String    m_string;
```

```
    }

    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String args[])
    {
        new Transaction("Create Object")
        {
            @Override
            protected void run() throws Rollback
            {
                new ShapeChange();
            }
        }.execute();
    }
}
```

On the second run, the following exception is thrown:

Example 14.2. Type change - second run output

```
[A] Java main class
                    com.kabira.snippets.reference.Shape.main exited with an exception.
[A] Java exception occurred: Audit of class
                    [com.kabira.snippets.reference.ShapeChange] failed:
                    Type did not match. New type name
                    com.kabira.snippets.reference.ShapeChange - existing type
                    name com.kabira.snippets.reference.ShapeChange.
                    Changed values :numberSlots:objectSize:;
                    the class will not be loaded.
[A] at com.kabira.platform.classloader.ClassLoader.createKTPTypeDescriptor
                   (Native Method)
[A] at com.kabira.platform.classloader.ClassLoader.defineManagedClass
                   (ClassLoader.java:642)
[A] at com.kabira.platform.classloader.ClassLoader.findClass(ClassLoader.java:302)
[A] at com.kabira.platform.classloader.ClassLoader.loadClass(ClassLoader.java:228)
[A] at java.lang.ClassLoader.loadClass(ClassLoader.java:251)
[A] at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:319)
[A] at com.kabira.snippets.reference.Shape.run(Shape.java:37)
[A] at com.kabira.platform.Transaction.execute(Transaction.java:132)
[A] at com.kabira.snippets.reference.Shape.main(Shape.java:31)
INFO: Application [com.kabira.snippets.reference.Shape6] running on node [A]
                   exited with status [-1]
INFO: Run of distributed application [com.kabira.snippets.reference.Shape6] complete.
```

Setting `reset = true` will avoid this exception.

Highly available and distributed systems When highly available or distributed Managed Objects are used in an application, the type definition for these classes are pushed to all nodes in a cluster. To ensure that the type definitions stay consistent on all nodes, the same value for the `reset` option must be sent to all nodes. This is accomplished using the Distributed Development features as described in Chapter 2.

Using inconsistent values for the `reset` option on nodes in a cluster will cause application failures because of the inconsistent type definitions on the nodes. The cluster must be restarted to resolve this problem.

Security

The TIBCO ActiveSpaces® Transactions deployment tool will prompt for a password if one is not specified on the command line, or in the `options` file, using the `password` parameter.

The TIBCO ActiveSpaces® Transactions deployment tool always uses SSL connections to TIBCO ActiveSpaces® Transactions nodes. This requires an SSL `TrustManagerFactory` algorithm that supports storage and management of X.509v3 credentials. By default the `SunX509` `TrustManagerFactory` is used. If that `TrustManagerFactory` algorithm is not supported in the JRE being used to execute the deployment tool, an alternative algorithm can be specified using the `ssl.TrustManagerFactory.algorithm` JVM system property.

The algorithm specified using the above property must reference a `TrustManagerFactory` which supports storage and management of X.509v3 credentials and is provided by the JRE being used.

Options file

When the TIBCO ActiveSpaces® Transactions deployment tool is executed it looks for the following file:

```
<user home directory>/.ast/options
```

If this file exists, any deployment tool command line options in the `options` file are used. Command line options specified in the `options` file have the same affect as the same command line option specified on the command line.

Options on the command line override the same option in the `options` file. For example if the command line contains `-jar deploy.jar debug=true` and the options file contains `debug = false`, a debug value of `true` is used for the application execution.

The options file must follow the format below.

```
#
#     Any line starting with '#' is ignored
#
#     Each option is specified on a separate line, as follows:
#
<option name> = <option value>[newline]
```

For example, the following `options` file would set up the default username and password for use with the TIBCO ActiveSpaces® Transactions development nodes:

```
#
#       Username and password for TIBCO ActiveSpaces® Transactions
 development nodes
#
username = guest
password = guest
```

Default JVM properties The options file also supports defining default JVM properties for the TIBCO ActiveSpaces® Transactions JVM. Default TIBCO ActiveSpaces® Transactions JVM properties are specified using the `jvmoptions` option name. Here is an example of using the `jvmoptions` option.

```
#
#       Username and password for TIBCO ActiveSpaces® Transactions
 development nodes
#
```

```
username = guest
password = guest

#
#       Default JVM options
#
jvmoptions = -Duser.name=fred -Duser.country=US
```

Deployment example

Example 14.3 on page 255 shows how to deploy a simple Java program to an TIBCO ActiveSpaces® Transactions node from the command line.

Example 14.3. Deployment example

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.reference;

/**
 * Simple hello world snippet
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class HelloWorld
{
    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String args[])
    {
        System.out.println("Hello World");
    }
}


#
#    NOTE: Remove the package statement from the HelloWorld.java snippet above
#    to work with the example steps below.
#

#
#    Compile the program using the native javac on the host machine
#
javac HelloWorld.java

#
#    Excute the program against an TIBCO ActiveSpaces® Transactions
 node - assumes deploy.jar is in local directory
*
java -jar deploy.jar hostname=192.168.1.128 adminport=2001 \
     username=guest password=guest HelloWorld

#
#    The output from the command
#
INFO: deploy.jar version: [core_linux081117]
INFO: node version: [core_linux081117]
Hello World
```

Upgrade utility

TIBCO ActiveSpaces® Transactions ships with an upgrade utility that is used during development to generate an *upgrade plan* used to deploy new versions of an application to TIBCO ActiveSpaces® Transactions nodes. See Chapter 8 for details on how the upgrade utility is used to upgrade applications.

The upgrade utility is executed from the command line. It is named `upgrade.jar`.

The general syntax for using the upgrade utility is:

```
java -jar upgrade.jar help  // display usage message
java -jar upgrade.jar [options]
```

`upgrade.jar` must be executed using the `-jar` option to a Java executable.

Table 14.2 on page 256 summarizes the supported upgrade utility options:

Table 14.2. Upgrade utility options

Option	Description
`current`	A `:` separated list of all directories where the current application JAR files can be found. This is a required option. **Note**: JVM-specific deployment directories are not automatically found by the upgrade utility. If they are being used, they must be explicitly listed.
`debug`	An enumeration indicating the level of diagnostic output. The valid values are `enable` - a high level description of the upgrade utility activity, `verbose` - detailed description of upgrade utility activity, or `none` - no diagnostic output. (default: `none`).
`upgradefile`	Optional `upgrade plan` file name. Default value is `upgrade.`*`<date>`* If the *upgrade plan* file already exists, a numeric value will be appended to make the file name unique.
`replacement`	A `:` separated list of directories where the replacement application JAR files can be found. This is a required option. **Note**: JVM-specific deployment directories are not automatically found by the upgrade utility. If they are being used, they must be explicitly listed.

The directories specified in the `current` and `replacement` options must contain all framework and application JAR files used by the application. The only exceptions are the directories containing the JDK and the TIBCO ActiveSpaces® Transactions SDK, i.e. `deploy.jar`, do not need to be specified. An error is reported by the upgrade utility if a class cannot be resolved. For example,

```
FAILURE: AUDIT: com/kabira/examples/chat/ActiveUser
Problem finding or parsing parent class com.kabira.businessstatemachine.Process
   in the current JAR files.
```

Supported JVM properties

In addition to the properties supported by the JVM, additional TIBCO ActiveSpaces® Transactions specific properties are defined in the the `Deploy` and `Status` classes in the `com.kabira.platform.property` package.

Debugging example

Example 14.4 on page 257 shows how to attach the Java debugger to an application running on TIBCO ActiveSpaces® Transactions .

Example 14.4. Debugging example

```
//     $Revision: 1.1.2.1 $

package com.kabira.snippets.reference;

/**
 * Simple hello world snippet
 * <p>
 * <h2> Target Nodes</h2>
 * <ul>
 * <li> <b>domainnode</b> = A
 * </ul>
 */
public class HelloWorld
{
    /**
     * Main entry point
     * @param args  Not used
     */
    public static void main(String args[])
    {
        System.out.println("Hello World");
    }
}


#
#    NOTE: Remove the package statement from the HelloWorld.java snippet above
#    to work with the example steps below.
#

#
#    Compile the program using the native javac on the host machine
#
javac HelloWorld.java

#
#    Excute the program against an TIBCO ActiveSpaces® Transactions
 node - assumes
#    deploy.jar is in the local directory
#    Notice that suspend=true is set on the command line to suspend main until the
#    debugger attaches.
#
#
java -jar deploy.jar hostname=192.168.1.128 adminport=2001 \
     username=guest password=guest suspend=true remotedebug=true HelloWorld

#
#    The output from the TIBCO ActiveSpaces® Transactions
 node - the above command
#    blocks waiting for the debugger to attach.
#
INFO: deploy.jar version: [core_linux081108]
INFO: JVM remote debugger agent running on [192.168.1.128:45871] ...
INFO: node version: [core_linux081108]
INFO: Running with suspend=true - the application will suspend execution before main()
is called ...
Listening for transport dt_socket at address: 45871
```

```
#
#    Execute jdb in another window using -connect to connect to the
#    TIBCO ActiveSpaces® Transactions
 node
#
#    NOTE: The port number is obtained from the above output.
#
jdb -connect com.sun.jdi.SocketAttach:hostname=192.168.1.128,port=45871

#
#    Output from jdb command - enter cont at the prompt
#
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
>
VM Started: No frames on the current call stack

main[1] cont

#
#    Output in jdb window after cont
#
>
The application exited


#
#    Output in java window after cont.
#
Hello World
```

Java Debug Wire Protocol

The Java Debug Wire Protocol (JDWP) is used to integrate debugging tools. Creating, reading, updating, and deleting managed objects is always done without a transaction. The implications of this are that modifications via JDWP clients may provide inconsistent transactional results in an active program and reading object data may display uncommitted data to the JDWP clients.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

X