Javlin User Guide

Release 6.1

February 2003

Javlin User Guide

ObjectStore Release 6.1 for all platforms, February 2003

The information in this manual is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear in this document.

The references in this manual to specific platforms supported are subject to change.

Contents

Preface . . . vii

Chapter 1 Javlin Overview . . . 1

Javlin Architecture . . . 2

Javlin Integration with J2EE Application Servers . . . 3

Basic Javlin Tasks . . . 4

Key Concepts . . . 4

Transactional Caching . . . 4

Javlin Caches . . . 5

Global and Virtual Transactions . . . 6

Operation and Global Contexts . . . 6

Access Mode . . . 7

Isolation Level . . . 8

Routing . . . 8

Scheduling . . . 9

Cache Storage . . . 9

Chapter 2 Declarative Configuration . . . 11

XML-Based Configuration File . . . 12

The Javlin Document Type Definition (DTD) . . . 12

External Entities in the Configuration File . . . 13

Names used by Javlin Deployment Descriptors . . . 14

Configuring The Cache Pool Manager . . . 15

XML Syntax . . . 15

Cache Pool Manager Descriptor Details . . . 15

Example. 16
Configuring Cache Pools 17
XML Syntax 17
Cache Pool Descriptor Details. 17
Example. 19
Configuring Cache Storage 20
XML Syntax 20
Database Descriptor Details 21
Root Descriptor Details 22
Object Descriptor Details 22
Example. 24
Configuring Deployed Components 25
XML Syntax 25
Method Descriptor Details 26
Example. 27
A Complete Example. 29

Chapter 3 Programmatic Configuration 33
Setting Up Transactional Caches. 33
Creating Databases 35
Creating and Retrieving Roots 35
Controlling Root Object Placement. 36
Working with MVCC Caches 37
Configuring a Cache Pool Manager for Multiple Caches. 38
Restrictions on Routing to an MVCC Cache. 38

Chapter 4 Virtual Transactions 41
Life Cycle of a Virtual Transaction 41
Beginning a Transaction 41
Committing a Transaction 42
Aborting a Transaction 42
Example of a Transaction. 43
Retrying a Transaction. 43
Classes to Use. 44

TransactionTry Methods 44
Using TransactionTryRuntimeExceptionsOnly 45
Using TransactionTryWrappedExceptions 45
Extending TransactionTry. 46
Programming Restrictions 47
Use of ObjectStore Java Classes 47
Retaining Persistent References Across Transactions. 47
Accessing Data Outside of a Transaction. 48
Thread Synchronization 49
Tips for Optimizing Transactions 49
Chapter 5 Routing and Scheduling 53
Routing Transactions 53
Initializing a Global Context 53
Routing to a Cache Pool 55
Routing to a Cache 55
Routing to an MVCC Cache 56
Routing Summary 57
Scheduling Transactions. 58
Group Open Interval 59
Commit If Idle. 60
Chapter 6 Configuring Javlin/JMTL 61
Configuring Javlin/JMTL Using Ant 61
Core Javlin/JMTL Configuration 62
Configuring Javlin/JMTL for Level Zero Integration 64
Configuring Javlin/JMTL for Level One Integration 65
Summary of Windows Environment Settings 66
Summary of UNIX Environment Settings. 67
Building and Deploying Applications 68
Building and Deploying on WebLogic 6.1 or 7.0 69
. . . . 69
Chapter 7 Using the Javlin Console 71
Starting the Javlin Console 71

Connecting to a Javlin Application 72
Disabling Javlin Monitoring 74
Overriding the Default Port Number 74
Configuring Data Displays 75
Data Panel 75
Graph Panel 80
Alerter Panel 81
Modifying Preferences 82
Saving and Reusing Console Settings 83
Stopping the Console 83
Index 85

Preface

The *Javlin User Guide* explains how to use Javlin and the Java Middle-Tier Library (JMTL) for Java applications that service large numbers of client requests. Javlin/JMTL can be used with or without application servers. This book also explains how to use Javlin/JMTL with Enterprise Java Beans (EJB) application servers.

How This Book is Organized

This book is organized as follows:

- Chapter 1, Javlin Overview, on page 1, provides a conceptual overview of Javlin and JMTL.
- Chapter 2, Declarative Configuration, on page 11, describes how to use XML-based deployment descriptors to configure Javlin.
- Chapter 3, Programmatic Configuration, on page 33, describes how to use the Javlin API to configure Javlin.
- Chapter 4, Virtual Transactions, on page 41, shows how to use the Javlin API to perform virtual transactions.
- Chapter 5, Routing and Scheduling, on page 53, explains how Javlin does the following:
    - Routes transactions to cache pools and caches.
    - Schedules transactions to minimize transaction-processing overhead.
- Chapter 6, Configuring Javlin/JMTL, on page 61, explains how to setup the Javlin/JMTL environment for building and deploying J2EE applications.
- Chapter 7, Using the Javlin Console, on page 71, explains how to use the Javlin Console to monitor transactional activity against Javlin caches.

Notation Conventions

This document uses the following conventions:

Convention	*Meaning*
`Monospace font`	`Monospace font` indicates code, syntax, system output, file names, and the like.
`Bold typewriter`	**`Bold typewriter font`** is used to emphasize particular code, such as user input.
`Italic typewriter`	*`Italic monospace font`* indicates the name of a variable for which you must supply a value. This typeface most often appears in a syntax line.
Sans serif	Sans serif typeface indicates the names of dialog boxes, buttons, fields, and so on, that are displayed in the user interface.
Italic serif	In text, *italic serif typeface* indicates the first use of an important term.
[ ]	Brackets enclose optional arguments.
{ *a* | *b* | *c* }	Braces enclose two or more items. You can specify only one of the enclosed items. Vertical bars represent OR separators. For example, you can specify *a* or *b* or *c*.
...	Three consecutive periods indicate that you can repeat the immediately previous item. In examples, they also indicate omissions.

ObjectStore on the World Wide Web

ObjectStore has its own Web site (www.objectstore.net) that provides a variety of useful information about products, news and events, special programs, support, and training opportunities.

Technical Support

When you purchase technical support, the following services are available to you:

- You can send questions to support@objectstore.net. Remember to include your site ID in the body of the electronic mail message.
- You can call the Technical Support organization to get help resolving problems.
- You can access the Technical Support Web site, which includes

- A template for submitting a support request. This helps you provide the necessary details, which speeds response time.
- Frequently asked questions (FAQs) that you can browse and query.
- Online documentation for all ObjectStore products.
- White papers and short articles about using ObjectStore products.
- Sample code and examples.
- The latest versions of ObjectStore products, service packs, and publicly available patches that you can download.
- Access to an ObjectStore product matrix.
- Support policies.
- Local phone numbers and hours when support personnel can be reached.

Education Services

Use the ObjectStore education services site (www.objectstore.net/services/education) to learn about the standard course offerings and custom workshops.

If you are in North America, you can call 1-800-477-6473 x4452 to register for classes. For information on current course offerings or pricing, send e-mail to classes@progress.com.

Searchable Documents

In addition to the online documentation that is included with your software distribution, the full set of product documentation is available on the ObjectStore Support Web server. The documentation is found at www.objectstore.net/documentation, and is listed by product. The site supports the most recent release and the previous supported release of ObjectStore documentation. Service Pack README files are also included to provide historical context for specific issues. Be sure to check this site for new information or documentation clarifications posted between releases.

Your Comments

ObjectStore product development welcomes your comments about its documentation. Send any product feedback to support@objectstore.net. To expedite your documentation feedback, begin the subject with `Doc:`. For example:

```
Subject: Doc: Incorrect message on page 76 of reference manual
```

Chapter 1
Javlin Overview

Javlin is designed for developing highly scalable applications that cache object data in the middle tier and service large numbers of client requests. Javlin can be used with any J2EE application server, including Enterprise Java Beans (EJB) servers/containers. Javlin can also be used without an application server. It is compatible with all J2EE APIs.

Javlin includes the Java Middle Tier Library (JMTL) which provides transparent, high-performance storage for Java objects. Javlin performs the following tasks automatically and transparently to your application:

- Caches persistent data accessed by client-initiated transactions
- Maintains the consistency and recoverability of the data caches
- Maintains each transaction's required isolation level
- Schedules transactions to optimize throughput

Javlin supports bean-managed persistence, as well as container-managed and bean-managed transactions.

This chapter covers the following topics:

Javlin Architecture 2

Javlin Integration with J2EE Application Servers 3

Basic Javlin Tasks 4

Key Concepts 4

Javlin Architecture

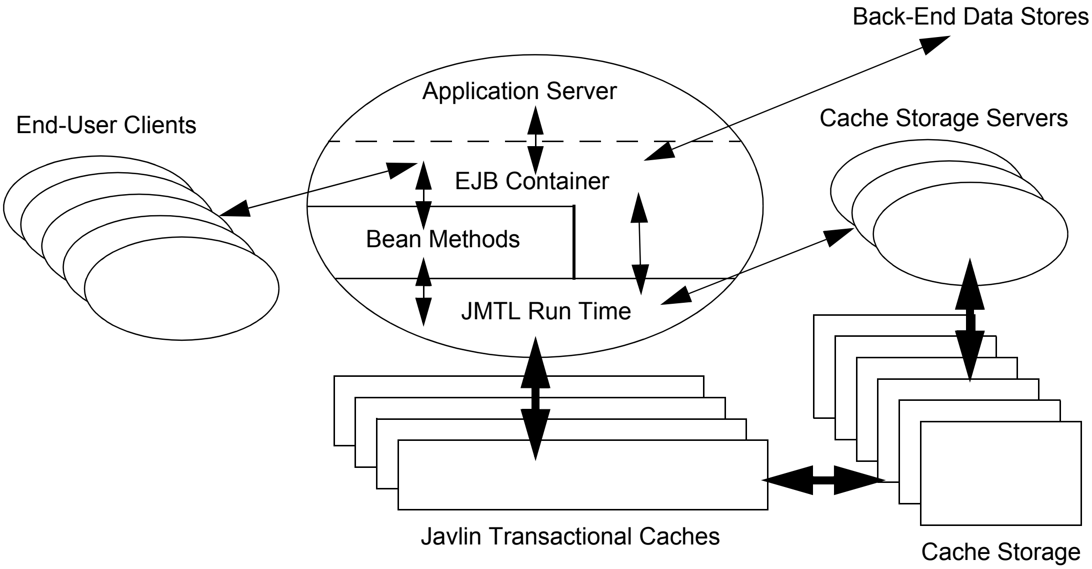
Javlin includes

- EJB container extensions, which integrate transactional caching with middleware application servers.
- JMTL (Java Middle-Tier Library), which provides a transactional-caching API.
- Database server and Java interface, which support cache synchronization, recoverability, and persistence.

Javlin Enterprise also includes database management support for Javlin databases, including such administrative facilities as

- Online backup and restore
- Replication
- Archive logging
- Failover

The following diagram illustrates Javlin's middle-tier role:



Javlin Integration with J2EE Application Servers

Javlin integrates transactional caching with J2EE-compliant application servers to support transaction coordination and persistence. Javlin supports two levels of integration which are described below.

Level Zero Integration

Level Zero integration can be used with or with out an application server. Component developers can use the Javlin API directly with any J2EE API (EJB, Servlets, JSP, etc.) However, Javlin transactions are not coordinated with the application server's transaction management (JTA) facility. For an EJB, this is equivalent to the Bean's transaction attribute being specified as `NotSupported`.

This is called Level Zero integration because no special support for the application server is provided by Javlin out of the box. Developers use Javlin's JMTL API with the stub version of Javlin's application server integration jar file.

Select Level 0 integration if you are not using any application server, or you are using an application server, but you do not require any integration of JMTL transactions with JTA transactions.

Level One Integration

Level One integration requires an application server supporting the Java Transaction Architecture (JTA). Component developers use the Javlin API directly with any J2EE API (EJB, Servlets, JSP, etc.) Javlin transactions are transparently coordinated by the application server's transaction management facility. Javlin participates as a *resource manager* within the application server's transaction management facility. Thus, Javlin transactions are integrated with the application server's globally-managed transaction context corresponding to any enclosing client-managed, bean-managed, or container-managed transaction.

A specific integration jar file is provided with Javlin for each application server with level one integration. Additionally, the build files and instructions for the example J2EE applications provided with Javlin include customization for application servers with level one integration support.

Javlin currently provides Level One integration with the BEA WebLogic Server.

Level Two Integration

In addition to transaction coordination, Level Two integration provides full container managed persistence (CMP) support for Entity Beans, so developers do not have to code directly to the Javlin API. The CMP integration is provided together with the Level One integration support in the application server specific integration JAR file.

Release 6.1 of Javlin does not currently provide this level of integration with any application server.

Basic Javlin Tasks

To implement Javlin transactional caching in your application, you perform the following basic tasks:

- Set up transactional caches
- Create Javlin cache storage (databases)
- Store and retrieve roots
- Perform virtual transactions

You perform these tasks using a combination of XML declarations in a deployment-descriptor file and Java programming in your application code, using the Javlin API.

When you use Javlin with Level Zero or Level One integration, you must perform some tasks programmatically, as described in Chapter 3, Programmatic Configuration, on page 33 and Chapter 4, Virtual Transactions, on page 41.

Key Concepts

The following concepts are central to understanding how to develop applications that use Javlin/JMTL.

Note

In the sections that follow, *italic font* indicates key concepts that are discussed elsewhere in this chapter.

Transactional Caching

Transactional caching is the underlying concept of Javlin. It refers to the automatic routing of client requests to separate Javlin caches, allowing inmemory access to cached objects. Javlin ensures data integrity by providing transactional consistency among the caches, even while servicing multiple, concurrent transactions against the same data. Each client request is handled as a separate Java thread executing against cached objects.

Javlin Caches

There are two types of Javlin caches:

- An *update* cache, which is used by transactions that require either write-access to data or read-only access to data that is completely up to date.
- An *MVCC* (Multiversion Concurrency Control) cache, which is used by read-only transactions that can rely on a recent snapshot of data.

Transactions that use an MVCC cache can perform nonblocking reads that allow concurrent transactions executing in update caches to read or write to the same data, without having to wait.

Note PSE Pro does not support MVCC caches.

Cache Pools

A cache pool consists of one or more caches and corresponds to a logical partitioning of data. Its main purpose is to make your application more scalable by allowing you to partition cached data into separate pools. If you have a very large data set, you can configure each pool to process logically related data. For example, you could configure different Cache Pools to process data from different geographical regions. Likewise, if your components perform many operations on the data set, you can configure the pools operationally, routing (for example) customer operations to one pool and administrative operations to another.

Configuring Javlin Caches

Configuring Javlin caches—specifying their organization, number, and size—is determined by a number of factors, including:

- The number, frequency, and duration of anticipated requests
- Whether or not a transaction is read-only (*access mode*)
- The extent to which a transaction needs to be isolated from concurrent transactions (*isolation level*)
- The extent to which data is logically partitioned

Declarative Configuration

Although you can use the Javlin API to configure Javlin's caches, the simplest way to configure them is declaratively, using XML-based deployment descriptors. The deployment descriptors specify information about the caches as well as about the component methods that will be executing transactions against the caches. Javlin processes the deployment descriptors and uses this information at run time to route transactions to caches.

The advantage of using deployment descriptors to configure Javlin is that you don't have to burden the application with configuration code. Deployment descriptors also allow you to re-configure or fine-tune Javlin's caches at deployment time without having to recompile your application.

The use of deployment descriptors to configure Javlin is described in Chapter 2, Declarative Configuration, on page 11.

Global and Virtual Transactions

The J2EE transaction model has two kinds of transactions:

- *Global transactions*. These correspond to JTA transactions managed and coordinated by a J2EE application server. These are managed by the `javax.transaction` APIs.
- *Local transactions*. These correspond to backend data source transactions. Local transactions are an implementation detail of either a component or its corresponding container.

In Javlin, a global transaction has the same meaning as in J2EE, but a local transaction is replaced by the concept of the *virtual transaction*. Virtual transactions are component-level transactions that occur whenever a component operation accesses persistent data. All access to persistent data must occur within the scope of a virtual transaction, as described in Chapter 4, Virtual Transactions, on page 41.

Operation and Global Contexts

Javlin uses information in two run-time structures to manage virtual transactions. These structures are:

- Operation Context, represented by the `OperationContext` class
- Global Context, represented by the `GlobalContext` class

The Operation Context encapsulates the following information about a virtual transaction:

- The name of the *Cache Pool* to which it should be routed
- Its *access mode*
- Its *isolation level*

Typically, this information is specified by the deployment descriptor for the component method. However, it can also be specified programmatically, using the Javlin API. When a virtual transaction starts to execute, this information becomes available to Javlin in the Operation Context that is used to begin the transaction.

When a global or independent transaction begins, Javlin creates a Global Context and associates it with the set of virtual transactions that are either controlled by the global transaction or nested in the independent transaction. (A virtual transaction is independent if it is not controlled by a global transaction.) Javlin initializes the Global Context based on the Operation Context of the *dominant transaction*.

Dominant transaction

To be dominant, a virtual transaction must meet the following conditions:

- It must be either the first virtual transaction in the global transaction or the outermost independent virtual transaction.
- It must be specified as dominant. Typically, dominance is specified by the `DominantOperation` attribute in the deployment descriptor for the component method, as described in Method Descriptor Details on page 26. However, it can also be specified programmatically, using the Javlin API.

Javlin uses the initialized Global Context to determine the *routing* of all virtual transactions in the global transaction or in a nest of independent transactions. All such virtual transactions must be compatible with the *access mode* and *isolation level* encapsulated in the Global Context. For example, if the access mode specified by the Global Context is read-only, then all nested virtual transactions must be read-only. If any nested transaction performs an update operation, the entire transaction aborts.

Access Mode

All operations on persistent data have an access mode of either read-only or update. The access mode is specified in the deployment-descriptor file and is used by Javlin at run time for *routing* and *scheduling* virtual transactions.

Isolation Level

Every transaction that accesses cached data has an isolation level that is typically specified in a deployment-descriptor file. The isolation level indicates how immune the transaction is to the effects of concurrent transactions on the same data. The weaker a transaction's isolation level, the more likely that it can execute concurrently with other transactions. Javlin uses the isolation level to optimize transaction processing when *routing* and *scheduling*.

The following table summarizes the isolation levels. The levels are arranged in order of increasing strictness, with the weakest level (`READ_UNCOMMITTED`) at the top. For more detailed information about isolation levels, see the *Javlin API Reference* for the `IsolationLevelConstants` interface.

Isolation Level	***Description***
`READ_UNCOMMITTED`	The transaction can commit even if it reads data that includes currently uncommitted changes made by other transactions. Read-uncommitted transactions can be routed to MVCC or update caches.
`READ_COMMITTED`	The transaction can read committed data only. Read-committed transactions can be routed to MVCC or update caches.
`REPEATABLE_READ`	The transaction can read committed data only. Also, the committed data cannot change during the course of the transaction. However, the reads do not have to be consistent with each other, as in the case of a transaction that is reading multiple databases. Repeatable-read transactions can be routed to MVCC or update caches.
`SERIALIZABLE`	The transaction can read committed data only. Also, the committed data cannot change during the course of the transaction, and the reads must be consistent with each other. Serializable transactions are *always* routed to update caches.

Routing

Routing is the process by which Javlin selects a *Cache Pool* and a cache within the Cache Pool in which a transaction will execute. Routing to a Cache Pool is based on the name of the Cache Pool as specified in an Operation Context. Routing to an update or MVCC cache is based on the transaction's *access* *mode* and *isolation level*. For example, a transaction with an access mode of update is always routed to an update cache.

For detailed information about how Javlin routes a transaction to a Cache Pool and cache, see Routing Transactions on page 53.

Scheduling

Javlin uses scheduling to reduce transaction overhead and interactions with external data sources. During a 500-ms interval (the default), Javlin schedules individual virtual transactions that have been routed to a cache to execute within a single, low-level physical transaction against *Cache Storage*. Once all currently scheduled transactions complete executing, Javlin performs a physical commit. At this point, all updated data is written to Cache Storage and becomes visible to other caches.

Javlin ensures transactional consistency within a physical transaction by scheduling only virtual transactions that are compatible with respect to their *isolation level* and *access mode*. For more detailed information about scheduling, see Scheduling Transactions on page 58.

Cache Storage

Javlin uses persistent Cache Storage to ensure durability for the cached data and to allow multiple caches to share the same data. Data is stored as objects within cache storage to reduce the cost of mapping data from a non-object data model to the object model being cached.

Cache Storage can be used either as the persistent backing store for Javlin's transactional caches or as the data source itself.

Chapter 2 Declarative Configuration

This chapter explains how to use XML-based deployment descriptors to configure Javlin. It covers the following topics:

XML-Based Configuration File 12

Configuring The Cache Pool Manager 15

Configuring Cache Pools 17

Configuring Cache Storage 20

Configuring Deployed Components 25

A Complete Example 29

Note

You must load configuration information from the deployment-descriptor files into an instance of the `com.odi.env.JVMEnvironment` class at run time, by calling the `deploy()` methods on the `JVMEnvironment` class; see the *Javlin API Reference*.

XML-Based Configuration File

To configure Javlin declaratively, you specify deployment descriptors in an XML-based configuration file. This section covers the following topics:

- Identifying Javlin's Document Type Definition (DTD), which Javlin uses to validate your configuration file.
- Referencing external entities in the configuration file.
- The names used by the Javlin deployment descriptors.

The Javlin Document Type Definition (DTD)

The Javlin deployment descriptors are specified in an XML-based file that conforms to Javlin's DTD. The DTD is specified in the `DOCTYPE` statement that must appear in the prologue to the XML file that contains your deployment descriptors as follows:

```
<!DOCTYPE JVMEnvironment SYSTEM "URI-specification">
```

This statement asserts that your XML file is compliant with the Javlin DTD specified by *`URI-specification`*. If the `DOCTYPE` statement appears in the prologue of your deployment-descriptor file, the Javlin parser will validate the file against the DTD; otherwise, the parser will not validate the file.

The Javlin DTD (`jmtl-dd.dtd`) is located in the JMTL `JAR` file. You specify its location in the `DOCTYPE` statement as follows:

```
<!DOCTYPE JVMEnvironment SYSTEM "JMTL:com/odi/jmtl/jmtl-dd.dtd">
```

The prefix `JMTL` that appears before the colon (`:`) is an extension to the XML syntax for the URI. This prefix causes the parser to use the following search rules for locating `com/odi/jmtl/jmtl-dd.dtd`:

DTD search rules

1. Get the `ClassLoader` for the component and invoke its `getResource()` method.
2. Get the `ClassLoader` for the system and invoke its `getResource()` method.
3. Use the `java.class.path` property.

If the parser cannot find the DTD according to these rules, a fatal error will occur and the configuration file will not be processed.

External Entities in the Configuration File

You can use XML’s `ENTITY` statement to insert an external entity in the XML file that contains your deployment descriptors. For example, instead of repeating the same deployment-descriptor information in different places in the XML file, you can do the following:

1 Create a file that contains the external entity.

2 Specify the external-entity file in the `ENTITY` statement.

3 Reference the external entity wherever you want it inserted in the XML file.

The `ENTITY` statement must appear in the `DOCTYPE` statement of the XML file. It has the following syntax:

```
<!ENTITY entity-reference SYSTEM "URI-specification">
```

The `JMTL` prefix can be used in *`URI-specification`* to cause the parser to use the same search rules to find the external entity that it uses to find the Javlin DTD; see The Javlin Document Type Definition (DTD) on page 12.

Use the `&`*`entity-reference`*`;` syntax to reference the entity in the XML file.

Example

Following is part of an example configuration file that illustrates the use of external entities:

```
<?xml version="1.0" ?>
<!DOCTYPE JVMEnvironment SYSTEM "JMTL:com/odi/jmtl/jmtl-dd.dtd"
                 [
               <!ENTITY dbandroots SYSTEM "tstdbmaps.xml">
               <!ENTITY employeepool SYSTEM
               "file:/os/dserve/java/mytest/emppool.xml">
               <!ENTITY customerspool SYSTEM
               "ftp://hodge:testXML@jeffrey.cats.com/custpool.xml">
                 ]>
               <JVMEnvironment>
                 <cache-pool-manager>
               <!-- define CachePool configurations for CachePoolManager -->
                   <cache-pools>

               <!-- use external entities for these shared pools -->
                      &employeepool;
                      &customerspool;
               <!-- define a new cache pool named AccountCachePool -->
                      <cache-pool name="AccountCachePool">
                        <NumberOfUpdateCaches>1</NumberOfUpdateCaches>
                        <NumberOfMvccCaches> 1 </NumberOfMvccCaches>
                        <AddressSpaceSize>0x8000000</AddressSpaceSize>
                        <CacheSize> 0x1000000 </CacheSize>
```

```
      </cache-pool>
    </cache-pools>
  </cache-pool-manager>

  <cache-storage>
<!-- use an external entity for the database and root config -->
  &dbandroots;
  </cache-storage>
  .
  .
  .
```

Names used by Javlin Deployment Descriptors

The names used by Javlin deployment descriptors correspond to Java qualified names that denote Javlin run-time structures within a Java Virtual Machine (JVM). The `<cache-pool-manager>` element is represented at run time by an instance of the `com.odi.jmtl.CachePoolManager` class. You can call methods on this object to set, retrieve, or modify its values. Likewise, you can set or modify the values in a deployment-descriptor file.

Using the XML-based deployment descriptors, you can configure each Javlin element. Descriptors that have attributes require you to specify literal values. Leading and trailing spaces are ignored. In the following example, the `<OverwriteDebugOutputFile>` attribute specifies that Javlin is to overwrite the debug file that it maintains:

```
<cache-pool-manager>
  . . .
  <OverwriteDebugOutputFile> true </OverwriteDebugOutputFile>
  . . .
</cache-pool-manager>
```

Other elements of Javlin are similarly configured, as described in the following sections.

Note This document uses the term *attribute* to refer to such names as `OverwriteDebugOutputFile` and `NumberOfMvccCaches`, which are used to configure the Cache Pool Manager, Cache Pools, etc. Javlin attributes are unrelated to XML attributes, which are used to provide information about XML elements.

Configuring The Cache Pool Manager

The Cache Pool Manager manages a set of Cache Pools within the JVM. The deployment-descriptor file specifies the following configuration information for the Cache Pool Manager:

- Attributes of the Cache Pool Manager
- The collection of Cache Pools

This information is encapsulated at run time in an instance of the `CachePoolManager` class, as described in the *Javlin API Reference*. There is only one Cache Pool Manager within a JVM. You can access this instance with the fully qualified name `com.odi.jmtl.CachePoolManager`.

XML Syntax

Use the following syntax to configure the Cache Pool Manager in an XML-based deployment-descriptor file:

```
<cache-pool-manager>
  <DebugLevel> string-value </DebugLevel>
  <DebugOutputFile> string-value </DebugOutputFile>
  <OverwriteDebugOutputFile>
    boolean-value
  </OverwriteDebugOutputFile>

  <!-- list of Cache Pool descriptors -->
  <cache-pools>
    . . .
  </cache-pools>
</cache-pool-manager>
```

See Configuring Cache Pools on page 17 for information about the Cache Pool descriptor.

Cache Pool Manager Descriptor Details

The Cache Pool Manager descriptor provides information about the management of Javlin's Cache Pools. Descriptor information must be enclosed within the `<cache-pool-manager>` and `</cache-pool-manager>` tags. The Cache Pool Manager descriptor has the following optional attributes:

Attributes

<DebugLevel> specifies the level of Javlin's output messages. If this attribute is specified, the value must be one of the following string literals:

Level	*Meaning*
NONE	No output
ERROR	Error information (default)
WARNING	Warning information, plus all information at the ERROR level
VERBOSE	Configuration changes, plus all information at the WARNING level
DEBUG	Routing and scheduling information, plus all information at the VERBOSE level

Note that the levels are accumulative; for example, if you specify VERBOSE, the output includes messages at the ERROR, WARNING, and VERBOSE levels.

<DebugOutputFile> specifies the path of the file that Javlin uses for writing output messages. The path must be expressed as a string literal and must be valid within the host operating system. If this attribute is not specified, Javlin writes output messages to System.out.

<OverwriteDebugOutputFile> specifies whether the Cache Pool Manager is to overwrite (true) or append (false) to an existing debug file. The value must be expressed as a Boolean literal. The default is to append (false).

Example

The following example configures a Cache Pool Manager that will output messages at level WARNING to debug.out, overwriting any information that was previously in the file.

```
<cache-pool-manager>
  <DebugLevel> WARNING </DebugLevel>
  <DebugOutputFile> debug.out </DebugOutputFile>
  <OverwriteDebugOutputFile> true </OverwriteDebugOutputFile>

  <!-- list of Cache Pool descriptors -->
  <cache-pools>
    . . .
  </cache-pools>
</cache-pool-manager>
```

Configuring Cache Pools

A Cache Pool comprises one or more caches and determines their number, type, size, and other characteristics. The deployment-descriptor file specifies the following configuration information for each Cache Pool:

- Name of the Cache Pool
- Cache Pool attributes
- The number and types of caches within a Cache Pool

This information is encapsulated at run time in an instance of the `CachePool` class, as described in the *Javlin API Reference*. A Cache Pool is denoted by `com.odi.jmtl.CachePoolManager.CachePools.`*`cache-pool-name`*, where *`cache-pool-name`* is the name you specify with the Cache Pool descriptor.

XML Syntax

Use the following syntax to configure a Cache Pool in an XML-based deployment-descriptor file:

```
<cache-pools>
  <cache-pool name="cache-pool-name">
    <NumberOfUpdateCaches>
      integer-value
    </NumberOfUpdateCaches>
    <NumberOfMvccCaches> integer-value </NumberOfMvccCaches>
    <AddressSpaceSize> integer-value </AddressSpaceSize>
    <CacheSize> integer-value </CacheSize>
    <UserTransactionUsage>string-value</UserTransactionUsage>
    <GroupOpenInterval> integer-value </GroupOpenInterval>
    <CommitIfIdle> boolean-value </CommitIfIdle>
    <GroupOpenIntervalMvcc>
      integer-value
    </GroupOpenIntervalMvcc>
    <CommitIfIdleMvcc> boolean-value </MvccCommitIfIdle>
  </cache-pool>
  . . .
</cache-pools>
```

Cache Pool Descriptor Details

The Cache Pool descriptor describes the Cache Pool whose name is *`cache-pool-name`*. A Cache Pool contains a logical set of Javlin caches. Cache Pool descriptor information must be enclosed within the following nested tags:

```
<cache-pools>
  <cache-pool name="cache-pool-name">
```

```
      . . .
   </cache-pool>
</cache-pools>
```

where *cache-pool-name* is a string literal that you must specify. This name is used in the fully qualified name of the Cache Pool, `com.odi.jmtl.CachePoolManager.CachePools.`*cache-pool-name*.

The Cache Pool descriptor has the following attributes:

Attributes

<AddressSpaceSize> specifies the number of bytes of persistent address space allocated for each cache in this Cache Pool. The number must be expressed as an unsigned integer literal, and must be greater than or equal to the number specified for `<CacheSize>`. The total amount of address space for all caches in the process must be less than or equal to the available address space on the current platform.

If `<AddressSpaceSize>` is not specified, Javlin uses no more than one cache, whose size will be the currently available persistent address space. For more information, see the `OS_AS_SIZE` and `OS_DEFAULT_AS_PARTITION_SIZE` environment variables in *Managing ObjectStore*.

<CacheSize> specifies the number of bytes of page caching allocated for each cache in this Cache Pool. The number must be expressed as an unsigned integer literal, and must be less than or equal to the number specified for `<AddressSpaceSize>`. If `<CacheSize>` is not specified, the default number of bytes is 8 MB (0x800000). For more information, see the `OS_CACHE_SIZE` environment variable in *Managing ObjectStore*.

<CommitIfIdle> specifies whether the scheduler for an update cache commits the physical transaction whenever no transactions are pending (`true`), or waits for `<GroupOpenInterval>` to elapse before committing the physical transaction (`false`). The value must be expressed as a Boolean literal. If this attribute is not unspecified, the default is `true`.

<CommitIfIdleMvcc> specifies whether the scheduler for an MVCC cache commits the physical transaction whenever no transactions are pending (`true`), or waits for `<GroupOpenIntervalMvcc>` to elapse before committing the physical transaction (`false`). The value must be expressed as a Boolean literal. If this attribute is not unspecified, the default is `false`.

<GroupOpenInterval> specifies the time interval in milliseconds during which the scheduler continues to schedule additional global and virtual transactions to execute in a physical transaction (transaction group). The value must be expressed as an unsigned integer literal. If this attribute is not unspecified, the default is `500`.

`<GroupOpenIntervalMvcc>` specifies the time interval in milliseconds during which the transaction scheduler for an MVCC cache continues to schedule additional global and virtual transactions to execute in a physical transaction (transaction group). The value must be expressed as an unsigned integer literal. If this attribute is not unspecified, the value of `<GroupOpenInterval>` is used.

`<LockTimeout>` specifies the number of milliseconds for a physical transaction to wait when attempting to acquire a database lock. If the lock cannot be acquired within the specified timeout, the phyiscal transaction aborts. This attribute applies only to update caches. The default is to wait until the lock becomes available.

`<MaxConcurrentTransactions>` specifies the maximum number of read-only transactions that are scheduled to run concurrently in a cache. This attribute applies to both update and MVCC caches. The default is 20.

`<NumberOfMvccCaches>` specifies the number of MVCC caches to create in this Cache Pool. The number must be expressed as an unsigned integer literal. If this attribute is not unspecified, the default is `0`.

`<NumberOfUpdateCaches>` specifies the number of update caches to create in this Cache Pool. The value must be expressed as an unsigned integer literal. If this attribute is not unspecified, the default is `1`.

`<UserTransactionUsage>` asserts usage information about the caches. The value must be expressed as a string literal and can be one of the following:

- `noUserTransactions`: asserts that this Cache Pool will never be used by a client- or component-initiated global transaction. This assertion allows Javlin to optimize use of the Cache Pool.
- `userTransactionsPossible`: asserts that this Cache Pool can be used by a client- or component-initiated global transaction. This assertion disables certain optimizations. This assertion is the default.

Example

The following example configures a Cache Pool named `TradingCachePool`. It will consist of two caches, one update and one MVCC. Each cache will have `0x8000000` bytes of available address space and `0x1000000` bytes for paging.

```
<cache-pools>
  <!-- define a Cache Pool named TradingCachePool -->
  <cache-pool name="TradingCachePool">
```

```
    <NumberOfUpdateCaches> 1 </NumberOfUpdateCaches>
    <NumberOfMvccCaches> 1 </NumberOfMvccCaches>
    <AddressSpaceSize> 0x8000000 </AddressSpaceSize>
    <CacheSize> 0x1000000 </CacheSize>
    <UserTransactionUsage>
      noUserTransactions
    </UserTransactionUsage>
  </cache-pool>
</cache-pools>
```

Configuring Cache Storage

Cache Storage is implemented as a set of object databases to provide object durability and distributed sharing of persistent objects. Javlin uses database roots to access objects in each database. The deployment-descriptor file configures the names and number of databases and roots.

Cache Storage information is encapsulated at run time in an instance of the `CacheStorage` class, as described in the *Javlin API Reference*. A database is denoted by the fully qualified name `com.odi.jmtl.CacheStorage.Databases.`*`logical-database-name`*, and a root by `com.odi.jmtl.CacheStorage.Roots.`*`logical-root-name`*. You specify *`logical-database-name`* and *`logical-root-name`* in the deployment-descriptor file, using the syntax described in the next section.

XML Syntax

Use the following syntax to configure Cache Storage in an XML-based deployment-descriptor file:

```
<cache-storage>
  <databases>
    <database-descriptor name="logical-database-name">
      <PhysicalName> string-value </PhysicalName>
    </database-descriptor>
    . . .
  </databases>

  <roots>
    <root-descriptor name="logical-root-name">
      <StorageName> string-value </StorageName>
      <PhysicalName> string-value </PhysicalName>
      <root-object-descriptor>
        <MethodSignature> string-value </MethodSignature>
        <ClassType> string-value </ClassType>
        <MethodArguments>
```

```
                <boolean-value> argument-value </boolean-value>
                <byte-value> argument-value </byte-value>
                <character-value> argument-value </character-value>
                <double-value> argument-value </double-value>
                <float-value> argument-value </float-value>
                <integer-value> argument-value </integer-value>
                <long-value> argument-value </long-value>
                <object-value>
                  object-descriptor-attributes
                </object-value>
                <short-value> argument-value </short-value>
                <string-value> argument-value </string-value>
              </MethodArguments>
            </root-object-descriptor>
          </root-descriptor>
          . . .
        </roots>
      </cache-storage>
```

See the following sections for information about the descriptors that you use to configure Cache Storage:

Database Descriptor Details 21

Root Descriptor Details 22

Object Descriptor Details 22

Database Descriptor Details

The database descriptor describes a database used by Javlin's cache storage. Descriptor information must be enclosed within the following nested tags:

```
<databases>
  <database-descriptor name="logical-database-name">
    . . .
  </database-descriptor>
</databases>
```

where *logical-database-name* is a string literal that you must specify as the logical name of the database you are configuring. This name maps to the actual name that you specify with the `<PhysicalName>` attribute; see below. You use the logical name when referring to the database in the deployment-descriptor file and in application code. The logical name is also used in the fully qualified name `com.odi.jmtl.CacheStorage.Databases.`*logical-database-name*.

The database descriptor has the following required attribute:

Attribute

`<PhysicalName>` specifies the path name of the database. The path name must be expressed as a string literal and must be valid within the host operating system. For information about database path names see Specifying File Database Pathnames in *Managing ObjectStore*.

Root Descriptor Details

The root descriptor describes a database root used by Javlin to retrieve objects in cache storage. Descriptor information must be enclosed within the following nested tags:

```
<roots>
  <root-descriptor name="logical-root-name">
    . . .
  </root-descriptor>
</roots>
```

where `logical-root-name` is a string literal that you must specify as the logical name of the root you are configuring. This name maps to the actual name that you specify with the `PhysicalName` attribute; see below. You use the logical name when referring to the root in the deployment-descriptor file and in application code. The logical name is also used in the fully qualified name `com.odi.jmtl.CacheStorage.Roots.logical-root-name`.

The root descriptor has the following attributes:

Attributes

`<PhysicalName>` specifies the actual name of the root. The name must be expressed as a string literal. This attribute is required.

`<root-object-descriptor>` describes the object used as the database root. This attribute is optional only if the `PhysicalName` attribute specifies the name of an existing root. For information about the attributes for the root object descriptor, see Object Descriptor Details on page 22.

`<StorageName>` specifies the logical name of the database that contains the root. The name must be expressed as a string literal and must be exactly the same as the name in one of the `<logical-database-name>` tags nested in the Cache Storage section of the deployment-descriptor file. This attribute is required.

Object Descriptor Details

The object descriptor describes how to construct or retrieve an object. It can be used to describe a database root or the object value of a method argument. When used as a root object descriptor, descriptor information must be enclosed within the following tags:

```
<root-object-descriptor>
  . . .
```

```
</root-object-descriptor>
```

When used to describe an object value, descriptor information must be enclosed within the following tags:

```
<object-value>
  . . .
</object-value>
```

The object descriptor has the following attributes:

Attributes

`<ClassType>` specifies the qualified name of the `java.lang.Class` instance that represents the type of the object. The name must be expressed as a string literal. This attribute is required.

`<MethodSignature>` specifies the signature of the constructor or the method that creates or retrieves the object. This attribute is optional. If this attribute is not specified, Javlin uses the no-argument constructor. If it is specified, the signature must be formatted as follows:

```
fully-qualified-class-name.method-name [ argument-list ]
```

where *`fully-qualified-class-name`* is the qualified name of the method's class type, *`method-name`* is the name of the method, and *`argument-list`* is the optional parenthesized argument list, consisting of comma-separated type names. The type names must be fully qualified for any types other than primitives. If *`argument-list`* is not specified, the method is assumed to have no arguments. If it is specified, you must also use the following `MethodArguments` attribute.

`<MethodArguments>` specifies the ordered list of the actual values that are passed to the method. This attribute is required only if the signature includes an argument list; see the `<MethodSignature>` attribute above.

With the exception of the `object-value` attribute, each of the following attributes specifies a value that must be expressed as a literal. The resulting value must conform to the Java type represented by the attribute name. The `object-value` attribute is an object descriptor that describes how to create or access an object, as described in this section:

- `<boolean-value>`
- `<byte-value>`
- `<character-value>`
- `<double-value>`
- `<float-value>`
- `<integer-value>`

- `<object-value>`
- `<long-value>`
- `<short-value>`
- `<string-value>`

Example

The following example configures Cache Storage to use one database (`TradingDB`) that has two roots (`Customers` and `Stocks`). Note that all three names are logical names. The fully qualified type name of the root objects is `com.odi.util.OSHashMap`.

```
<cache-storage>
  <databases>
    <database-descriptor name="TradingDB">
      <PhysicalName>
        ${rootPath}/trading/data/trading.odb
      </PhysicalName>
    </database-descriptor>
  </databases>

  <roots>

    <root-descriptor name="Stocks">  <!-- logical root name -->
      <StorageName> TradingDB </StorageName>
      <PhysicalName> trading.stocks </PhysicalName>
      <root-object-descriptor>
        <MethodSignature>
          com.odi.util.OSHashMap(int)
        </MethodSignature>
        <ClassType> com.odi.util.OSHashMap </ClassType>
        <MethodArguments>
          <integer-value>100</integer-value>
        </MethodArguments>
      </root-object-descriptor>
    </root-descriptor>

    <root-descriptor name="Customers">
      <StorageName> TradingDB </StorageName>
      <PhysicalName> trading.customers </PhysicalName>
      <root-object-descriptor>
        <ClassType> com.odi.util.OSHashMap </ClassType>
      </root-object-descriptor>
    </root-descriptor>

    <root-descriptor name="Stocks">
      <StorageName> TradingDB </StorageName>
      <PhysicalName> trading.stocks </PhysicalName>
      <root-object-descriptor>
        <ClassType> com.odi.util.OSHashMap </ClassType>
      </root-object-descriptor>
    </root-descriptor>
  </roots>
</cache-storage>
```

Configuring Deployed Components

The component section of the deployment-descriptor file describes each component that is deployed within a JVM and uses Javlin. This section includes information about each component method that performs transactions on cached data.

XML Syntax

Use the following syntax to configure a component in an XML-based deployment-descriptor file:

```
<component-descriptor name="fully-qualified-component-name">

  <method name="method-signature">
    <method-descriptor>
      <CachePoolName> string-value </CachePoolName>
      <Update> boolean-value </Update>
      <IsolationLevel> string-value </IsolationLevel>
      <TransactionAttribute>
        string-value
      </TransactionAttribute>
      <DominantOperation> boolean-value </DominantOperation>
      <RetryLimit> integer-value </RetryLimit>
    </method-descriptor>
  </method>
  . . .
</component-descriptor>
```

The `<component-descriptor>` tag must specify the `name` attribute, where *`fully-qualified-component-name`* is a string literal that represents the fully qualified name for the component. The `<component-descriptor>` tag can only be used to enclose `<method>` tags, as well as any tags that they enclose.

Instead of the `<component-descriptor>` tag, you can nest tags for each element of the fully qualified name; see the example at the end of the XML file in Example on page 27. Note that you *must* use the nested-tag syntax if you want to enclose any application data that is not specific to Javlin.

Method Descriptor Details

The method descriptor describes a component method that performs transactions on cached data. Descriptor information must be enclosed within the nested tags:

```
<method name="method-signature">
  <method-descriptor>
  . . .
  </method-descriptor>
</method>
```

Signature syntax

where `method-signature` is a string literal that has the following format:

```
method-name [ ( argument-list ) ]
```

where `method-name` is the name of the method and `argument-list` is an optional list of comma-separated type names. The type names must be fully qualified for any types other than primitives. If `argument-list` is not specified, the method is assumed to have no arguments.

Overloadings and default methods

Methods having the same name can share (or overload) the same method descriptor. You can also define a default method descriptor for all component methods that are not explicitly defined. To define a default method, specify the string `default` for the signature, as in the following:

```
<method name="default">
  . . .
</method>
```

The method descriptor has the following optional attributes:

Attributes

<CachePoolName> specifies the name of the Cache Pool to which the method should be routed. The name must be expressed as a string literal. This attribute is optional. If unspecified, Javlin arbitrarily selects a Cache Pool for the method.

<Update> specifies the access mode of the transaction performed by the method as either update (`true`) or read-only (`false`). The value must be expressed as a Boolean literal. This attribute is optional. If unspecified, the access mode is update (`true`). For more information, see Access Mode on page 7.

<IsolationLevel> specifies the degree to which the method is immune to the effects of concurrent transactions on the same data. The value must be expressed as a string literal and can be one of the following:

- `SERIALIZABLE` (default)

- `REPEATABLE_READ`
- `READ_COMMITTED`
- `READ_UNCOMMITTED`

For information about these values, see Isolation Level on page 8.

`<TransactionAttribute>` specifies the global (JTA) transaction requirements of the method. The value must be expressed as a string literal and can be one of the following:

- `Never`
- `Mandatory`
- `NotSupported`
- `Required`
- `RequiresNew`
- `Supports` (default)

Javlin does not initiate global transactions. The value specified for `<TransactionAttribute>` influences transaction routing. For more information about the influence of `<TransactionAttribute>` on routing, see Initializing a Global Context on page 53.

Note

If you are using an EJB application server, the transaction attribute that you specify in the standard EJB deployment descriptor must be the same as the transaction attribute that you specify in the Javlin deployment descriptor.

`<DominantOperation>` specifies whether or not this method can be dominant. The value must be expressed as a Boolean literal. If this attribute is not specified, the method is not dominant (`false`). For more information about dominant transactions, see Operation and Global Contexts on page 6.

`<RetryLimit>` specifies the maximum number of times that the transaction performed by the method will be retried following an abort. This attribute applies only to transactions initiated by `TransactionTry.execute()`. The value must be expressed as an unsigned integer literal. If unspecified, the default is 4 (`TransactionTry.getRetryLimitDefault()`).

Example

The following descriptor example configures the deployed component `TraderBean`. It defines four methods:

- `getCustomers()`

- getStocks()
- getPrices()
- getPositions()

All four methods perform read-only transactions that are routed to the Cache Pool named TradingCachePool. They all have an isolation level of READ_COMMITTED, and their global transaction requirement is Required. None of the methods have arguments.

All other component methods default to performing update transactions that are routed to the Cache Pool named TradingCachePool.

```
<component-descriptor
name="com.odi.examples.trading.TraderBean">
  <method name="getCustomers">
    <method-descriptor>
      <CachePoolName> TradingCachePool </CachePoolName>
      <Update> false </Update>
      <IsolationLevel> READ_COMMITTED </IsolationLevel>
      <TransactionAttribute>
        Supports
      </TransactionAttribute>
    </method-descriptor>
  </method>

  <method name="getStocks">
    <method-descriptor>
      <CachePoolName> TradingCachePool </CachePoolName>
      <Update> false </Update>
      <IsolationLevel> READ_COMMITTED </IsolationLevel>
      <TransactionAttribute>
        Supports
      </TransactionAttribute>
    </method-descriptor>
  </method>

  <method name="getPrices">
    <method-descriptor>
      <CachePoolName> TradingCachePool </CachePoolName>
      <Update> false </Update>
      <IsolationLevel> READ_COMMITTED </IsolationLevel>
      <TransactionAttribute>
        Supports
      </TransactionAttribute>
    </method-descriptor>
  </method>

  <method name="getPositions">
    <method-descriptor>
```

```
      <CachePoolName> TradingCachePool </CachePoolName>
      <Update> false </Update>
      <IsolationLevel> READ_COMMITTED </IsolationLevel>
      <TransactionAttribute>
        Supports
      </TransactionAttribute>
    </method-descriptor>
  </method>

  <method name="default">
    <method-descriptor>
      <CachePoolName> TradingCachePool </CachePoolName>
      <Update> true </Update>
    </method-descriptor>
  </method>
</component-descriptor>
```

A Complete Example

The following is an example of a deployment-descriptor file that configures Javlin’s caches:

```
<?xml version="1.0"?>
<!DOCTYPE JVMEnvironment SYSTEM "JMTL:com/odi/jmtl/jmtl-dd.dtd">
<JVMEnvironment>
<!-- Configuration for CachePoolManager -->
  <cache-pool-manager>
    <DebugLevel> WARNING </DebugLevel>
    <DebugOutputFile> debug.out </DebugOutputFile>
    <OverwriteDebugOutputFile> true </OverwritedebugOutputFile>
    <cache-pools>
      <cache-pool name="TradingStateful">
        <NumberOfUpdateCaches> 1 </NumberOfUpdateCaches>
        <NumberOfMvccCaches> 1 </NumberOfMvccCaches>
        <AddressSpaceSize> 0x8000000 </AddressSpaceSize>
        <CacheSize> 0x1000000 </CacheSize>
      </cache-pool>
    </cache-pools>
  </cache-pool-manager>

  <cache-storage>
    <databases>
      <database-descriptor name="TradingDB">
        <PhysicalName>
          ${rootPath}/trading/data/trading.odb
        </PhysicalName>
      </database-descriptor>
    </databases>

    <roots>
      <root-descriptor name="Customers">
        <StorageName> TradingDB </StorageName>
        <PhysicalName> trading.customers </PhysicalName>
        <root-object-descriptor>
          <ClassType> com.odi.util.OSHashMap </ClassType>
        </root-object-descriptor>
      </root-descriptor>

      <root-descriptor name="Stocks">
        <StorageName> TradingDB </StorageName>
        <PhysicalName> trading.stocks </PhysicalName>
        <root-object-descriptor>
          <ClassType> com.odi.util.OSHashMap </ClassType>
        </root-object-descriptor>
      </root-descriptor>
    </roots>
  </cache-storage>

  <component-descriptor name="com.odi.examples.trading.TraderBean">
    <method name="getCustomers">
      <method-descriptor>
        <CachePoolName> TradingStateful </CachePoolName>
        <Update> false </Update>
        <IsolationLevel> READ_COMMITTED </IsolationLevel>
        <TransactionAttribute> Supports </TransactionAttribute>
      </method-descriptor>
    </method>

    <method name="getStocks">
      <method-descriptor>
        <CachePoolName> TradingStateful </CachePoolName>
        <Update> false </Update>
        <IsolationLevel> READ_COMMITTED </IsolationLevel>
        <TransactionAttribute> Supports </TransactionAttribute>
      </method-descriptor>
    </method>

    <method name="getPrices">
      <method-descriptor>
        <CachePoolName> TradingStateful </CachePoolName>
        <Update> false </Update>
        <IsolationLevel> READ_COMMITTED </IsolationLevel>
        <TransactionAttribute> Supports </TransactionAttribute>
      </method-descriptor>
    </method>
    <method name="getPositions">
      <method-descriptor>
        <CachePoolName> TradingStateful </CachePoolName>
        <Update> false </Update>
        <IsolationLevel> READ_COMMITTED </IsolationLevel>
        <TransactionAttribute> Supports </TransactionAttribute>
      </method-descriptor>
    </method>

<!-- The following represents the fully qualified default method name -->
<!-- "com.odi.examples.trading.TraderBean.default" -->
    <method name="default">
      <method-descriptor>
        <CachePoolName> TradingStateful </CachePoolName>
        <Update> true </Update>
      </method-descriptor>
    </method>
  </component-descriptor>

<!-- The following are application-defined names used by the component. -->
  <StockPrices>
    <WEBL> 10.0 </WEBL>
    <INTL> 15.0 </INTL>
  </StockPrices>
  <tradeLimit> 500 </tradeLimit>
  <customers> ${rootPath}/trading/data/customers.txt </customers>
  <stocks> ${rootPath}/trading/data/stocks.txt </stocks>
  <database> TradingDB </database>
</JVMEnvironment>
```

Chapter 3 Programmatic Configuration

This chapter explains how to use the API to configure Javlin. It covers the following topics:

Setting Up Transactional Caches 33

Creating Databases 35

Creating and Retrieving Roots 35

Working with MVCC Caches

Note For information about the Javlin API for using virtual transactions, see Chapter 4, Virtual Transactions, on page 41.

Setting Up Transactional Caches

The simplest way to set up transactional caches is in an XML-based deployment-descriptor file. For more information about using deployment descriptors to configure transactional caches, see Chapter 2, Declarative Configuration, on page 11. If your application requires you to set up caches programmatically, you can use the Javlin API. Configuring Javlin programmatically is more difficult to implement, makes your application harder to maintain, and should be used only if you cannot use deployment descriptors.

To set up a single `Cache`, you create a `CachePoolManager` containing a single `CachePool`. Follow these steps:

1 Create a `CachePoolConfiguration` with a class constructor, passing a `String`, the name you want to give to the `CachePool`.

2 Create a `Map` with an entry for each database that the cache will use. Each entry maps a logical database name to a database path.

3 For each database root your application uses, create a `RootMapping`, which encapsulates a logical database name, a physical root name, and the name of a class.

4 Create a `Map` with an entry for each root. Each entry maps a logical root name to an instance of `RootMapping`.

5 Create a `CachePoolManagerConfiguration` with the class constructor, passing the `CachePoolConfiguration` and the `Map`s created in the previous steps.

6 Create a `CachePoolManager` with `new` and the class constructor, passing the `CachePoolManagerConfiguration` created in the previous step.

The following example sets up Javlin caches:

```
// Create a Cache Pool configuration object
CachePoolConfiguration cpConfig =
  new CachePoolConfiguration("cachePool1");

// Create dbNameMap for Cache Pool Manager configuration
Map dbNameMap = new HashMap();
dbNameMap.put("TraderDB", "/foo/bar/trading/db1.odb");

// Create a root mapping
RootMapping rootMapping =
  new RootMapping(
    "TraderDB", "root1", "com.odi.util.OSHashMap");

// Create a logical root map
Map rootMap = new HashMap();
rootMap.put("TraderRoot", rootMapping);

// Create a Cache Pool Manager configuration object
CachePoolManagerConfiguration cpmConfig =
  new CachePoolManagerConfiguration(
    cpConfig,
    null, // partitions
    dbNameMap,
    rootMap,
    null // attribute map
  );
  // Create cache pool manager
  cachePoolManager = new CachePoolManager(cpmConfig);
```

Creating Databases

To use Javlin to store and retrieve persistent data, you must create one or more Javlin databases or use existing ObjectStore or PSE Pro databases. In addition, you must create at least one database root in each database.

To create a database with Javlin, follow these steps:

1. Use `CachePoolManager.getCachePool()` to retrieve a `CachePool` that will use the database, passing in the name of the `CachePool` that you want to retrieve.
2. Use `CachePool.createDatabase()` to create a database, passing the logical name of the database you want to create and the new database's access mode. Make sure that the name you pass is already defined in the `CachePoolManager`'s map of logical database names.

The following example creates a database:

```
cachePoolManager.getCachePool("cachePool1").createDatabase(
  "TraderDB",
  ObjectStore.ALL_READ | ObjectStore.ALL_WRITE
);
```

By specifying a logical database name as shown in this example, you ensure that the code will remain stable even if the database's path changes. If you move a database, you only have to change an entry in the `CachePoolManager`'s database-name `Map`. Use `CachePoolManager.setDBMapping().`

Creating and Retrieving Roots

To create or retrieve a database root, you must be within a virtual transaction; see Chapter 4, Virtual Transactions, on page 41.

To create a database root, do the following:

1. Use the static method `Cache.getCurrent()` to retrieve the cache in which the current transaction is executing.

2 Use `Cache.getRootValue()` to create or retrieve the object associated with a given logical root name. Make sure that the name is already defined in the cache pool manager's map of logical root names.

The following example creates and retrieves a root:

```
Map traderRootMap =
   (Map) Cache.getCurrent().getRootValue("foo");
```

The first time you call `getRootValue()` on a given root name, the method creates an instance of the initialization class specified in the root's `RootMapping`. It uses the class's no-argument constructor and then returns the new object. Subsequent calls to `getRootValue()` retrieve the object and return it. The root object and all objects reachable from it are stored in the database specified by the root's `RootMapping`.

Databases and roots are discussed in detail in the ObjectStore documentation. See, for example, the *ObjectStore Java Tutorial*.

Controlling Root Object Placement

By default, Javlin places all root objects, and all objects reachable from them, in the default database segment and cluster. You can improve the performance of your application by grouping together objects that are used together, and conversely separating objects that are never used together.

To control the database placement of root objects, you can provide a custom root object constructor or factory class. For example, the following factory class creates a new cluster in the default segment and allocates the root object in that cluster:

```
public class OSTreeSetFactory {

   public static OSTreeSet create(String logicalDBName) {
      Cache cache = Cache.getCurrent();
      Database db = cache.getDatabase(logicalDBName);

      // Create new cluster in the default segment
      Cluster cluster = db.getDefaultSegment().createCluster();

      // Place new root object in the new cluster
      OSTreeSet treeset = new OSTreeSet(cluster);
      return treeset;
   }
}
```

In order for Javlin to invoke this factory method when constructing the root object, you must specify the method name and argument values in the `<root-object-descriptor>` as shown in the following example:

```
<root-descriptor name="CustomerRoot">
  <StorageName> BankDB </StorageName>
  <PhysicalName> CustomerBean.root </PhysicalName>
  <root-object-descriptor>
    <MethodSignature>
      com.acme.util.OSTreeFactory.create(java.lang.String)
    </MethodSignature>
    <ClassType> com.odi.util.OSTreeSet </ClassType>
    <MethodArguments>
      <string-value> BankDB <string-value>
    </MethodArguments>
  </root-object-descriptor>
</root-descriptor>
```

The first time you call `Cache.getRootValue("CustomerRoot")`, Javlin calls the specified method to construct the root object. The new `OSTreeSet` is allocated in the newly created cluster, and all objects subsequently added to the `OSTreeSet` will be allocated in the same cluster by default.

For more information about using database segments and clusters, see Grouping Objects in Multiple Segments and Clusters in the *ObjectStore Java API User Guide*.

Working with MVCC Caches

For read-only operations that do not require a completely up-to-date view of persistent data, you can increase throughput by using an MVCC cache. For information about MVCC caches, see Javlin Caches on page 5.

To use an MVCC cache, you must do the following:

- Include an MVCC cache in a `CachePool` when configuring the `CachePoolManager`. See the next section, Configuring a Cache Pool Manager for Multiple Caches.
- Use Operation Context to indicate whether a given transaction should be routed to an MVCC cache. See Restrictions on Routing to an MVCC Cache on page 38.

Configuring a Cache Pool Manager for Multiple Caches

To include an MVCC cache in a `CachePool`, pass an attribute map to the `CachePoolConfiguration` constructor. Each of the map's entries maps a `String` (an attribute name) to a `String` (the attribute's value). Include an entry that specifies the number of MVCC caches you want in the Cache Pool (the `NumberOfMvccCaches` attribute), as well as an entry that specifies the number of non-MVCC caches (the `NumberOfUpdateCaches` attribute).

Here is an example of how to configure the Cache Pool Manager to have one Cache Pool that contains one MVCC cache and one update cache:

```
// Create attribute map for Cache Pool configuration
Map cachePoolAttributes = new HashMap();

// Set attribute values
cachePoolAttributes.put("NumberOfUpdateCaches", "1");
cachePoolAttributes.put("NumberOfMvccCaches", "1");

// Create a Cache Pool configuration object
CachePoolConfiguration cpConfig =
  new CachePoolConfiguration(
    "cachePool1",
    cachePoolAttributes
  );
```

After you have configured for multiple caches, configuring the Cache Pool Manager is the same as when configuring for a single cache. See steps 2 - 6 and the example in Setting Up Transactional Caches on page 33.

Restrictions on Routing to an MVCC Cache

A virtual transaction is normally routed to an MVCC cache if its Operation Context specifies both of the following conditions:

- The transaction is read-only.
- The transaction has an isolation level of `REPEATABLE_READ` or weaker; see Isolation Level on page 8 for a table that lists and describes the isolation levels. As the table shows, an isolation level of `SERIALIZABLE` is stronger than `REPEATABLE_READ`, and therefore a transaction with an isolation level of `SERIALIZABLE` is *not* routed to an MVCC cache.

Note

If a virtual transaction is controlled by a global transaction or is nested in an virtual transaction, it may be routed to an update cache, depending on the context. For detailed information about routing, see Chapter 5, Routing and Scheduling, on page 53.

Chapter 4 Virtual Transactions

This chapter shows how to use the Javlin API to perform virtual transactions. It includes the following sections:

Life Cycle of a Virtual Transaction 41

Retrying a Transaction 43

Programming Restrictions 47

Tips for Optimizing Transactions 49

Life Cycle of a Virtual Transaction

A component method cannot access cached, persistent data outside the dynamic boundaries of a virtual transaction. The following sections explain how to use the Javlin API to define a virtual transaction.

Note Starting a middleware global transaction does not start a virtual transaction. A global transaction serves to group together virtual transactions that participate in a single, top-level request.

Also, a virtual transaction has a single thread associated with it for its entire duration. Other threads cannot join the virtual transaction.

Beginning a Transaction

To begin a virtual transaction, you call `VirtualTransaction.begin()`. This method returns a newly initiated virtual transaction.

`VirtualTransaction.begin()` takes an `OperationContext` object as an argument. As explained in Operation and Global Contexts on page 6, Javlin uses information encapsulated in an `OperationContext` object to route and schedule the new transaction. To create the object, call `BasicOperationContext.create()`, passing the bean's class and the signature of the method that performs the virtual transaction. Here is an example:

```
OperationContext operationContext =
  BasicOperationContext.create(
    Order.class, "getCreditCardInfo");
```

Committing a Transaction

A transaction completes successfully when it commits its changes to cached data, making them permanent and visible to other transactions. To commit a transaction, call `commit()`.

When you call `commit()` on a virtual transaction that is controlled by a global transaction or nested in another virtual transaction, the virtual transaction's changes do not immediately become permanent in Cache Storage and therefore are not visible to other caches and transactions. Instead, its changes become permanent and visible when either the controlling global transaction or the top-level virtual transaction commits.

When you call `commit()` on a virtual transaction that is neither controlled by a global transaction nor contained within another virtual transaction, the virtual transaction's changes become permanent and visible to other transactions upon completion of the `commit()`.

Note When a transaction commits, you can no longer access cached data without starting another transaction.

Aborting a Transaction

A transaction terminates unsuccessfully when it aborts. When a transaction aborts, all changes to cached data are undone and do not become permanent and visible in the database. Persistent data is rolled back to its state prior to the beginning of the virtual transaction.

To abort a transaction, call `VirtualTransaction.abort()`.

When a virtual transaction aborts:

- If the aborting transaction is controlled by a global transaction, then the global transaction also aborts.
- If the aborting transaction is routed to an update cache, then all virtual transactions that are grouped with it in the same *physical* transaction also

abort. If the aborting transaction is routed to an MVCC cache, any virtual transactions in the same physical transaction do not abort. For information about physical transactions, see Scheduling Transactions on page 58.

When you handle exceptions thrown during a virtual transaction, you can determine if the transaction is active with `VirtualTransaction.isActive()`, and abort it if necessary.

Example of a Transaction

Following is an example of a virtual transaction:

```
// Create an OperationContext
OperationContext operationContext =
  BasicOperationContext.create(
    Performance.class, "viewSeats");

try {
  // Start a virtual transaction
  transaction = VirtualTransaction.begin(operationContext);

  // Perform the operation
  availableSeatsGraphic = performance.viewSeats(
    theaterName, performanceData);

  // Commit the virtual transaction
  transaction.commit();
} catch (Exception e) {
  throw new ProcessingErrorException("txn error: " + e);
}
finally {
  // If the transaction is still active, abort it
  if (transaction != null && transaction.isActive()) {
    transaction.abort();
  }
}
```

Retrying a Transaction

When a virtual transaction aborts, Javlin signals `VirtualTransaction.RetryException` or `GlobalTransaction.AbortAndRetryException`. The exception can be thrown at any time during the virtual transaction.

Your code should handle these exceptions and retry the aborted virtual transaction. One way to do this is to define a class with an abstract method for performing a transaction's business logic, and a nonabstract method for executing and retrying the transaction.

Classes to Use

The Javlin API provides the following classes for retrying a transaction:

- `TransactionTry`. The base class of all `TransactionTry` extensions. Extend this class to define a customized retry class. Typically, this is done to account for particular checked exceptions declared by a component method.
- `TransactionTryRuntimeExceptionsOnly`. This class derives from `TransactionTry`. Use this class for transactions whose business logic cannot throw checked exceptions (that is, exceptions that must be listed in the `throw` clause of methods that signal them).
- `TransactionTryWrappedExceptions`. This class also derives from `TransactionTry`. Use this class for transactions whose business logic can throw checked exceptions.

TransactionTry Methods

Use the following methods to retry a transaction. They are defined in the base class `TransactionTry`:

- `execute()`: This method tries a virtual transaction and continues to retry it (up to a retry limit) if necessary. Within a retry loop, this method starts a virtual transaction by using the specified context, calls `action()`, and commits the virtual transaction. It returns the object returned by `action()` for the successful try. This method takes an `OperationContext` argument.
- `action()`: This method is an abstract method that is implemented by the anonymous inner class that extends one of the default `TransactionTry` classes. The `action()` method performs the behavior to be tried and retried, if necessary. This method is called by `execute()`. The `action()` method does not include virtual transaction demarcation; `execute()` starts and ends a virtual transaction for each try. This method returns a result `Object`.

The following procedure explains how to use these methods to try (and automatically retry, if necessary) a virtual transaction. You can use them with either TransactionTryRuntimeExceptionsOnly or TransactionTryWrappedExceptions:

1. Instantiate the class by doing the following:
   a. Use new and a constructor.
   b. Implement action().
2. Call execute() on the instance, passing the OperationContext object you want.

Using TransactionTryRuntimeExceptionsOnly

Use this class for transactions whose business logic throws only unchecked run-time exceptions. Calling execute() on an instance of this class tries to execute a virtual transaction, and automatically retries it if necessary. Run-time exceptions signaled by the virtual transaction's business logic are rethrown by execute().

Here is an example:

```
TransactionTryRuntimeExceptionsOnly transactionTry
  = new TransactionTryRuntimeExceptionsOnly() {
    public Object action() { /* business logic */ }
  }
OperationContext operationContext =
  BasicOperationContext.create( /* class and method-name */ );
transactionTry.execute(operationContext);
```

Using TransactionTryWrappedExceptions

Use this class for transactions whose business logic can throw checked exceptions. As with TransactionTryRuntimeExceptionsOnly, calling execute() on an instance of this class tries to execute a virtual transaction, and automatically retries it if necessary. However, with this class, exceptions signaled by the virtual transaction's business logic are wrapped by TransactionTryException, which is signaled by execute().

Here is an example that uses TransactionTryWrappedExceptions to handle the user-defined exception XE:

```
TransactionTryWrappedExceptions transactionTry
  = new TransactionTryWrappedExceptions() {
    public Object action() throws XE {
    /* business logic */; throw new XE() }
  }
OperationContext operationContext =
  BasicOperationContext.create( /* qualified method name */ );
```

```
try {
  transactionTry.execute(operationContext);
} catch (TransactionTryException tte) {
/* Handle wrapped exception. */ }
```

Extending TransactionTry

Here is an example of a class that allows direct handling of a checked exception, `CustomException`:

```
/*
  The class TransactionTryRemoteExceptions is used to simplify
  the use of virtual transaction using actions declared to
  throw RemoteExceptions.
*/
public abstract class TransactionTryRemoteExceptions
    extends TransactionTry {
  // Public Construction:
  /*
    Construct a TransactionTryRemoteExceptions object suitable
    to execute a virtual transaction using actions declared to
    throw RemoteExceptions with the default number of retries.
  */
  public TransactionTryRemoteExceptions ()
  {
    super();
  }
  // Protected Actions:
  // action() to be executed within a virtual transaction;
  // Object is the result of the action.
  protected abstract Object action ()
    throws RemoteException;

  // execute() will execute a virtual transaction within a retry
  // loop, passing it an instance of OperationContext
  public Object execute(OperationContext operationContext)
    throws RemoteException
  {
    Object result = null;
    try {
      result = super.execute(operationContext);
    } catch (RuntimeException runtimeException) {
      throw runtimeException;
    } catch (RemoteException remoteException) {
      throw remoteException;
    } catch (Exception exception) {
      // Cannot happen given overriding definition of action()
      Assert.conditionWithExplanation(
        false, "Unexpected exception: " + exception);
    }
    // Return the result of this execution.
```

```
        return result;
    }
}
```

Programming Restrictions

This section describes programming restrictions applying to applications using Javlin virtual transactions.

Use of ObjectStore Java Classes

Javlin layers on top of the ObjectStore Java Interface (OSJI). The OSJI classes `com.odi.Session` and `com.odi.Transaction` should not be used in a Javlin application. Higher-level classes, including `com.odi.jmtl.Cache` and `com.odi.jmtl.VirtualTransaction` are used in their place.

OSJI Interface	***Javlin Equivalent***
`com.odi.Session`	`com.odi.jmtl.Cache`
`com.odi.Transaction`	`com.odi.jmtl.VirtualTransaction` `com.odi.jmtl.util.TransactionTry`

Javlin applications can and should use all of the other OSJI classes, such as the persistence-capable versions of the Java 2 collections (`com.odi.util`) and the query facility (`com.odi.util.query.Query`). For additional information on using the OSJI classes, see the *ObjectStore Java API User Guide*.

Retaining Persistent References Across Transactions

Javlin uses the transaction retain mode of `ObjectStore.RETAIN_HOLLOW`, allowing applications to retain references to persistent objects across virtual transactions. In order to use this feature, the application needs to keep track of which cache a reference is associated with, since a method may not always route to the same cache instance. If a method references a persistent object in a cache other than the one in which it was initially retrieved, `com.odi.WrongSessionException` is thrown.

To avoid the `WrongSessionException` exception, the application can create a transient table (`java.util.Map`, for example) of references to persistent objects keyed by cache name. Alternatively, the application can access a root object in the current cache and then re-navigate to the desired object.

Note

All ObjectStore applications need to be careful about retaining references to non-exported objects across transactions, as these objects can be deleted or moved by intervening transactions. References to root objects, and other exported objects, are 'delete safe' and thus it is always safe to use them across transactions.

For additional information see the following sections in Chapter 6 of the Java API User Guide:

- Making Persistent Objects Hollow
- Caution About Retaining Unexported Objects

Accessing Data Outside of a Transaction

Threads in Javlin applications can only access persistent objects when associated with a virtual transaction. When a thread is not in a virtual transaction, Javlin disassociates the calling thread from the cache (and the underlying ObjectStore session). If the thread references a persistent object outside of a transaction, the exception `com.odi.NoSessionException` is thrown.

If the application needs access to persistent data outside a virtual transaction, the application must make a transient copy of the required data. The `com.odi.jmtl.util.TransactionTry` classes will automatically make a transient copy of objects returned from the `TransactionTry.action()` method. A transient copy is made if two conditions are met:

- The returned object is persistent.
- The returned object is serializable.

When these conditions are met, the `TransactionTry` classes call `ObjectStore.deepFetch()` on the returned object, and then serialize and de-serialize the resulting tree of objects to produce a transient copy. The transient copy is then returned to your application by `TransactionTry.execute()`.

Using `TransactionTry` classes to automatically make transient copies of data can result in returning more data than is needed by the application. Alternatively, the application can create its own transient copies in the `TransactionTry.action()` method and return just the data that is needed.

Thread Synchronization

As recommended for EJB applications, Javlin applications should not use application-controlled thread synchronization (synchronized methods or objects). If an application uses thread synchronization, thread deadlocking can result. For example, thread deadlocking results from the following scenario:

- "ThreadA" calls a method that routes to "Cache1" and is scheduled to run in a transaction group
- "ThreadB" calls a *synchronized* method that routes to "Cache1", but it arrives after `GroupOpenInterval` elapses so it blocks, waiting for the next transaction group
- "ThreadA" attempts to call the same synchronized method as "ThreadB", causing "ThreadA" to block and preventing "Cache1" from committing and starting a new transaction group

ThreadA and ThreadB are deadlocked since each is locking a resource that the other thread needs. ObjectStore cannot detect such a deadlock since it involves transient Java resources.   Thread deadlocking can be avoided if application-controlled thread synchronization is used only when Javlin is not active in the application, for example, before the `CachePoolManager` is initialized or after it is shutdown.

Tips for Optimizing Transactions

One of the design principles of Javlin is to minimize the effects of transaction processing on performance. Javlin's transactional caching architecture realizes this goal in the following ways:

- By routing virtual transactions to execute within separate caches, Javlin promotes concurrent execution of the transactions.
- By scheduling multiple virtual transactions that have been routed to a cache to execute within a single, low-level physical transaction, Javlin reduces the commit overhead of writing data to Cache Storage.
- By providing separate caches for update transactions and MVCC read-only transactions, Javlin minimizes the blocking effects that update transactions can have on other transactions.

When coding virtual transactions in your applications, you can use the following techniques to take full advantage of Javlin's transactional caching architecture:

- **Reduce the time spent in update transactions.**

  Update transactions are more costly in terms of performance than MVCC read-only transactions because they block concurrent transactions on the same data. And the longer they take to execute, the more costly they are. By splitting a long update transaction into several transactions, you open up intervals for the concurrent execution of other transactions, thus increasing throughput.

- **Route long-running transactions that execute frequently to a separate cache from short-running transactions.**

  As explained in Scheduling on page 9, Javlin routes multiple virtual transactions to a cache and then schedules them to execute in a single physical transaction every 500 ms. After the interval expires and all the scheduled virtual transactions commit, the physical transaction commits. However, if a virtual transaction takes longer than the 500-ms interval to execute, no other virtual transactions can be scheduled in this physical transaction. In effect, they are blocked for at least as long as it takes for the long-running transaction to commit.

  However, if you route long-running transactions to a separate cache, you allow Javlin to schedule more short-running transactions to execute within the same physical transaction, without being blocked by long-running transactions.

- **If your application has several related, consecutively executing update transactions that operate on the same data, nest them in a global or independent transaction.**

  As explained in Scheduling Transactions on page 58, a top-level update transaction must wait for the commit of the physical transaction before it can commit. (A *top-level* transaction is a virtual transaction that is neither controlled by a global transaction nor nested in another virtual transaction.) By nesting the update transactions in a top-level transaction, you allow them to commit their changes without having to wait for the physical transaction to commit. This technique allows multiple update transactions to be scheduled in a single physical transaction, thus reducing the overhead of committing a physical transaction.

  For information about the relationship between virtual and physical transactions, see Scheduling on page 9.

- **Route read-only transactions to an MVCC read-only cache.**

  An MVCC read-only cache is essentially a non-blocking read-only cache. Requests that are routed to an MVCC cache are allowed to read a snapshot of data that may be getting updated concurrently in a different update cache. MVCC requests do not require read locks on data that would block other requests from simultaneously updating the data. MVCC thus allows much higher overall throughput in the presence of updates. An MVCC cache's "snapshot" of data is constantly kept up to date by Javlin, which regularly commits the underlying physical transaction in the MVCC cache.

- **Use as few caches as necessary.**

  Do not add caches if your application's workload does not warrant it. Adding more caches increases the overhead of transaction processing. Two virtual transactions can execute faster if routed to a single cache and scheduled in a single physical transaction than if routed to separate caches and scheduled in two physical transactions.

- **Use only one update cache per logical data partition**.

  When multiple update caches access the same data, lock contention and deadlocks can occur. Deadlocks result in costly transaction aborts and retries, affecting all transactions currently scheduled in an update cache. To avoid deadlocks, use only one update cache per Cache Pool, and do not access the same data in more than one Cache Pool having an update cache.

Note that none of these techniques for optimizing transactions will compromise the consistency of Javlin's transactional caches.

Chapter 5 Routing and Scheduling

When application code begins a virtual transaction, Javlin routes the transaction, first to a Cache Pool, and then to a cache within the Cache Pool. After routing, Javlin schedules the transaction to execute with other, compatible virtual transactions in a physical transaction.

The following sections explain how Javlin routes and schedules transactions.

Routing Transactions

Javlin manages every virtual transaction by executing it within a cache. Given that you can configure multiple Cache Pools and multiple caches within each Cache Pool, it is important to understand how Javlin selects a cache for a virtual transaction, especially when you want to optimize transaction performance by routing to an MVCC cache.

The following sections explain how Javlin uses Global Context to route a virtual transaction to a Cache Pool and, once inside a Cache Pool, to a cache.

Initializing a Global Context

As explained in Operation and Global Contexts on page 6, every virtual transaction is associated with both an Operation Context and a Global Context. The Operation Context is specific to the transaction, and the Global Context is associated with a group of one or more virtual transactions that are either in a global transaction or nested inside an independent transaction. Javlin uses information encapsulated in the Global Context to route each virtual transaction in the group.

The following sections describe how Javlin creates and initializes a Global Context for the following:

- A global transaction
- An independent virtual transaction and any virtual transactions nested within it. (An independent virtual transaction is not controlled by a global transaction.)

Global transaction

When the first virtual transaction participates in a global transaction, Javlin creates an instance of `GlobalContext`. It uses the first virtual transaction's `OperationContext` to initialize the `GlobalContext` if any one of the following conditions is true:

- This virtual transaction is also *dominant*. For information about using the `DominantOperation` attribute to specify that a virtual transaction can be dominant, see Method Descriptor Details on page 26.
- The `OperationContext` specifies that this virtual transaction's `TransactionAttribute` is set to `RequiresNew` or `NotSupported`. For information about the `TransactionAttribute` attribute, see Method Descriptor Details on page 26.
- The `OperationContext` specifies a Cache Pool that has its `UserTransactionUsage` attribute set to `noUserTransactions`. For information about the `UserTransactionUsage` attribute, see Cache Pool Descriptor Details on page 17.

If none of these conditions is true, Javlin initializes the instance of `GlobalContext` with an isolation level of `SERIALIZABLE` and an access mode of Update.

After initializing `GlobalContext`, Javlin associates it with each virtual transaction that participates in the global transaction.

Independent transaction

If a virtual transaction does not participate in a global transaction and is outermost (that is, is not nested in another independent transaction), Javlin uses its `OperationContext` to initialize the `GlobalContext`. After initializing `GlobalContext`, Javlin associates it with the outermost transaction and any nested transactions it contains.

Note

It is the user's responsibility to ensure that `GlobalContext` settings will not result in incorrect behavior. For example, if the `GlobalContext` is set to an access mode of read-only and isolation level of `REPEATABLE_READ`, all subsequent virtual transactions controlled by the same global transaction must have compatible access modes and isolation levels. Javlin throws the exception `GlobalTransaction.IncompatibleContextException` when an `OperationContext` is incompatible with the current `GlobalContext`.

Routing to a Cache Pool

Javlin routes a virtual transaction to the Cache Pool that is named in the transaction's Operation Context if either of the following mutually exclusive conditions is true:

- Javlin used this transaction's Operation Context to initialize the Global Context; that is, this transaction is either dominant in a global transaction or is the outermost independent transaction. For information about the Global Context and how it is initialized, see Initializing a Global Context on page 53.
- Javlin did *not* use this transaction's Operation Context to initialize the Global Context, but the Global Context that is associated with this transaction specifies an access mode of read-only and an isolation level of `READ_COMMITTED` or `READ_UNCOMMITTED`.

In either of these cases, Javlin routes the transaction to the Cache Pool named in the transaction's Operation Context. If a Cache Pool with the specified name does not exist, Javlin throws an exception. If the Operation Context does not specify a Cache Pool name, Javlin arbitrarily selects a Cache Pool.

In all other cases, Javlin ignores the Cache Pool name specified in the Operation Context for this transaction and routes it to the Cache Pool that was selected for the previous transaction associated with the same Global Context. Note that virtual transactions that are either controlled by a Global or nested in another virtual transaction are always routed to the same Cache Pool (and cache instance) if the isolation level specified in the Global Context is `REPEATABLE_READ` or stricter.

Routing to a Cache

Once Javlin has routed a transaction to a Cache Pool, it routes the transaction to a cache within the Cache Pool as follows:

- If this transaction's Operation Context initialized the Global Context, then Javlin selects a cache according to the access mode and isolation level in the Operation Context, as follows:
  - If the access mode is update or the isolation level is `SERIALIZABLE`, the transaction is routed to an update cache.

  - If the access mode is read-only and the isolation level is `REPEATABLE_READ` or weaker, the transaction is routed to an MVCC cache.

- If Javlin did *not* use this transaction's Operation Context to initialize the Global Context *and* the Global Context that is associated with this transaction specifies an access mode of read-only and an isolation level of `READ_COMMITTED` or `READ_UNCOMMITTED`, then the transaction is routed as follows:

  - If the transaction's Operation Context specifies the name of a Cache Pool that was previously selected for a transaction that is associated with the same Global Context, the transaction is routed to the same cache within the Cache Pool.
  - If the transaction's Operation Context specifies the name of a different Cache Pool, the transaction is routed to an MVCC cache within that Cache Pool.

In all other cases, Javlin ignores the Operation Context for this transaction and routes it to the cache that was selected for the previous transaction associated with the same Global Context.

For more information about a transaction's access mode and isolation level, see Access Mode on page 7 and Isolation Level on page 8. Both are typically specified in the deployment descriptor for the component method, as described in Configuring Deployed Components on page 25.

Routing to an MVCC Cache

You can ensure that a read-only virtual transaction is always routed to an MVCC cache by doing the following:

1 Set its isolation level to a value less strict than `SERIALIZABLE`. For information about isolation levels, see Isolation Level on page 8. For information about using the `IsolationLevel` attribute, see Method Descriptor Details on page 26.

2 Make sure that the Cache Pool specified for this transaction has its `NumberOfMvccCaches` attribute set to at least `1`. For more information about this attribute, see Cache Pool Descriptor Details on page 17.

3 Make sure that this transaction's `OperationContext` is used to initialize the `GlobalContext`. For more information, see Initializing a Global Context on page 53.

Routing Summary

For transactions that are not controlled by a global transaction, and transactions that are dominant, the routing rules are summarized as follows:

Independent or dominant virtual transactions	***Route to Cache Pool specified in:***	***Route to Cache type:***
Update `SERIALIZABLE`	Operation Context	Update
Read-only `SERIALIZABLE`	Operation Context	Update
`REPEATABLE_READ`	Operation Context	MVCC
`READ_COMMITTED`	Operation Context	MVCC
`READ_UNCOMMITTED`	Operation Context	MVCC

For transactions that are controlled by a global transaction, or are nested in an outermost independent transaction, the routing rules are summarized as follows:

Dependent virtual transaction	***Routes to Cache Pool specified in:***	***Routes to Cache type:***
Update `SERIALIZABLE`	Global Context	Update
Read-only `SERIALIZABLE`	Global Context	Update
`REPEATABLE_READ`	Global Context	Update or MVCC depending on the Global Context
`READ_COMMITTED`	Operation Context	Latest cache used, or MVCC cache if routed to a different Cache Pool
`READ_UNCOMMITTED`	Operation Context	Latest cache used, or MVCC cache if routed to a different Cache Pool

Scheduling Transactions

Once a virtual transaction is routed to a cache, Javlin schedules it to execute with other virtual transactions in a single, low-level physical transaction against Cache Storage. To ensure that concurrent execution does not result in conflicting behavior among the virtual transactions in the same physical transaction, the transactions must all be compatible with respect to their isolation level and access mode. For example, a read-only transaction can never conflict with another read-only transaction, and Javlin can always schedule read-only transactions to execute concurrently—up to the maximum allowable number.

On the other hand, update transactions can never execute concurrently with any other virtual transaction—regardless of its access mode. Read-only transactions can execute concurrently with other read-only transactions only if their isolation levels are compatible; see Isolation Level on page 8

A virtual transaction commit will return control to the calling program before the commit of the physical transaction in which it is scheduled if any of the following conditions are true:

- The virtual transaction is contained in a global transaction or is nested in another virtual transaction.
- The virtual transaction is read-only, has an isolation level that is more strict than `READ_UNCOMMITTED`, and no update transaction has run in the cache.
- The virtual transaction is read-only and has an isolation level of `READ_UNCOMMITTED`.

	Run concurrently with other transactions	*Top-level transactions return to caller before the physical commit*
Update `SERIALIZABLE`	No	No
Read-only `SERIALIZABLE`	Yes	If no update has run
`REPEATABLE_READ`	Yes	If no update has run
`READ_COMMITTED`	Yes	If no update has run
`READ_UNCOMMITTED`	Yes	Yes

The commit of a virtual transaction must wait for the commit of its physical transaction if both of the following conditions are true:

- The virtual transaction is not controlled by a global transaction and is not nested in another virtual transaction.
- The virtual transaction is either an update transaction or is a read-only transaction with an isolation level that is more strict than `READ_UNCOMMITTED` and runs in a cache where an update transaction has run.

Group Open Interval

Javlin continues to schedule virtual transactions to execute in the same physical transaction until the time interval defined by either `GroupOpenInterval` or `GroupOpenIntervalMvcc` elapses. `GroupOpenInterval` is a Cache Pool attribute that defaults to 500 milliseconds and applies to update caches. `GroupOpenIntervalMvcc` is a Cache Pool attribute that defaults to 500 milliseconds and applies to MVCC caches.

After `GroupOpenInterval` (or `GroupOpenIntervalMvcc`) elapses, Javlin waits until all currently scheduled virtual and global transactions request a commit, and then performs the physical transaction commit. At this point, all updated data is written to Cache Storage and becomes visible to other caches.

Long running virtual or global transactions can delay the physical transaction commit, and can result in a queue of transactions waiting to be scheduled. To prevent long running transactions from monopolizing a cache, use your application server's transaction timeout facility to automatically abort long running global transactions.

There is generally no penalty to setting `GroupOpenIntervalMvcc` to a larger value, since read-only transactions routed to an MVCC cache return to the caller before the physical transaction commit. However, setting `GroupOpenInterval` to a larger value can increase the duration of top-level update transactions, since these transactions must wait for the physical transaction commit before returning to the caller.

Use caution when setting `GroupOpenInterval` to a smaller value as this can reduce overall system performance since each virtual transaction may then incur the overhead of a physical transaction commit. Instead, consider setting the `CommitIfIdle` attribute to `true`.

For information about using the GroupOpenInterval and GroupOpenIntervalMvcc attributes to control the duration of a physical transaction, see Cache Pool Descriptor Details on page 17.

Commit If Idle

Javlin will commit the physical transaction before GroupOpenInterval elapses if all of the following conditions are met:

- All currently scheduled transactions have requested a commit.
- There are no transactions routed to this cache and waiting to be scheduled.
- The cache is an update cache and CommitIfIdle is set to true, or the cache is an MVCC cache and CommitIfIdleMvcc is set to true .

CommitIfIdle is a Cache Pool attribute that defaults to true. This setting allows transactions executing in an update cache to return to the calling program faster when the cache is idle. However, a setting of true can reduce the overall system performance since each virtual transaction may then incur the overhead of a physical transaction commit.

CommitIfIdleMvcc is a Cache Pool attribute that defaults to false. Since transactions routed to an MVCC cache always return to the calling program before the physical commit, there is no benefit to committing the physical transaction early.

For information about using the CommitIfIdle and CommitIfIdleMvcc attributes to control the duration of a physical transaction, see Cache Pool Descriptor Details on page 17.

Chapter 6 Configuring Javlin/JMTL

This chapter describes the steps required to configure your environment for building and deploying Javlin/JMTL applications for both UNIX and Windows.

This chapter covers the following topics:

Configuring Javlin/JMTL Using Ant on page 61

Configuring Javlin/JMTL for Level Zero Integration on page 64

Configuring Javlin/JMTL for Level One Integration on page 65

Summary of Windows Environment Settings on page 66

Summary of UNIX Environment Settings on page 67

Building and Deploying Applications on page 68

Configuring Javlin/JMTL Using Ant

The Javlin examples use *Apache Ant* build files to automate the configuration, deployment, building and running of Javlin applications. Ant is a portable, Java-based build tool, similar to *make*.

You can use the example Ant build files to configure, build, deploy and run your own Javlin applications by customizing settings in the Ant build files. The Ant build files include extensions for many popular application servers. For more information on using the Ant build files see the `README` file in the Javlin `examples` directory.

If you choose not to use Ant, then follow the instructions in this chapter to manually configure your Javlin build and runtime environments.

Core Javlin/JMTL Configuration

Follow these steps to configure your environment to Javlin with or without an application server.

1 Set the `JMTL`, `OSJI` and `OS_ROOTDIR` environment variables to specify the installation root directories for the *Javlin*, *OSJI* and *ObjectStore*, respectively. For example:

On Windows:

```
set JMTL=c:\odi\javlin
set OSJI=c:\odi\osji
set OS_ROOTDIR=c:\odi\ostore
```

On Unix:

```
setenv JMTL /opt/ODI/javlin
setenv OSJI /opt/ODI/osji
setenv OS_ROOTDIR /opt/ODI/ostore
```

Note: The `JMTL` and `OSJI` variables names are not required names, they are merely used as shorthand when setting `CLASSPATH` and `PATH` in subsequent steps. The `OS_ROOTDIR` variable name is required by the core ObjectStore product.

2 If you are using Javlin on a UNIX platform, add the following directories to your shared library environment variable (`LD_LIBRARY_PATH,` `LIBPATH,` or `SHLIB_PATH`):

UNIX Shared Library Path Additions
`$OS_ROOTDIR/lib` `$OSJI/lib`

These are the ObjectStore and OSJI shared libraries on UNIX.

3 Add the following directories to your `PATH` environment variable:

	PATH Additions

Windows	`%JMTL%\bin`
	`%OS_ROOTDIR%\bin`
	`%OSJI%\bin`

UNIX	`$JMTL/bin`
	`$OS_ROOTDIR/bin`
	`$OSJI/bin`

Note: Your `PATH` variable must also contain the standard entries for Java development (that is, settings so that standard tools like *javac* and *rmic* can be executed).

4 Add the following `JAR` files to your `CLASSPATH` environment variable:

	CLASSPATH Additions

Windows	`%JTML%\xerces.jar`
	`%JMTL%\jmtl.jar`
	`%OSJI%\osji.jar`
	`%OSJI%\tools.jar`

UNIX	`$JMTL/xerces.jar`
	`$JMTL/jmtl.jar`
	`$OSJI/osji.jar`
	`$OSJI/tools.jar`

5 On HP-UX 32 bit platforms and on Solaris, you need to specify the environment variable `LD_PRELOAD` to ensure that the C++ delete operators are loaded in the correct order.

On Solaris:

```
set LD_PRELOAD=libosopdel.so
```

On HP-UX:

```
set LD_PRELOAD=libos.sl:libosth.sl
```

Configuring Javlin/JMTL for Level Zero Integration

Javlin's Level Zero integration provides no special support for the application server. For more information on Level Zero integration, see Javlin Integration with J2EE Application Servers3.

To configure your environment for Level Zero integration:

1 Set you environment as described in Core Javlin/JMTL Configuration on page 62.

2 Add the stub version of Javlin's application server integration jar file to your `CLASSPATH` environment variable:

	CLASSPATH Addition
Windows	`%JMTL%\jmtlnoappserver.jar`
UNIX	`$JMTL/jmtlnoappserver.jar`

Configuring Javlin/JMTL for Level One Integration

Javlin's Level One integration provides integration of Javlin transactions with the application server's JTA transactions. For more information on Level One integration, see Javlin Integration with J2EE Application Servers on page 3.

To configure your environment for Level One integration:

1 Set your environment as described in Core Javlin/JMTL Configuration on page 62.

2 Add one of the following Javlin application server integration jar files to your `CLASSPATH` environment variable. Select the jar file corresponding to the application server you are using:

Application Server	*CLASSPATH addition*
BEA WebLogic 6.1	`jmtlwl61.jar`
BEA WebLogic 7.0	`jmtlwl70.jar`

The `JAR` files are located in the Javlin installation directory (`%JMTL%` on Windows or `$JMTL` on UNIX). Select one of these `JAR` files and add it to your `CLASSPATH`, in addition to the core `JAR` files.

Summary of Windows Environment Settings

Environment Variable	*Description and Setting*
`OS_ROOTDIR`	Required by ObjectStore. Set to the ObjectStore installation directory.
`OSJI`	Shorthand for the ObjectStore Java installation directory.
`JMTL`	Shorthand for the Javlin installation directory.

`PATH`	Required to locate tools and shared libraries. Setting must include:
	`%JMTL%\bin`
	`%OS_ROOTDIR%\bin`
	`%OSJI%\bin`

`CLASSPATH`	Required to locate ObjectStore and Javlin classes. Setting must include:
	`%JMTL%\xerces.jar`
	`%JMTL%\jmtl.jar`
	`%OSJI%\osji.jar`
	`%OSJI%\tools.jar`
	For *Level Zero* integration, or when not using any application server, add the following:
	`%JMTL%\jmtlnoappserver.jar`
	For *Level One* integration, add one of the following, depending on the application server you use:
	`%JMTL%\jmtlwl61.jar`
	`%JMTL%\jmtlwl70.jar`

Summary of UNIX Environment Settings

Environment Variable	*Description and Setting*
`OS_ROOTDIR`	Required by ObjectStore. Set to the ObjectStore installation directory.
`OSJI`	Shorthand for the ObjectStore Java installation directory.
`JMTL`	Shorthand for the Javlin installation directory.

`PATH`	Required to locate tools. Setting must include:
	`$JMTL/bin`
	`$OS_ROOTDIR/bin`
	`$OSJI/bin`

`CLASSPATH`	Required to locate ObjectStore and Javlin classes. Setting must include:
	`$JMTL/xerces.jar`
	`$JMTL/jmtl.jar`
	`$OSJI/osji.jar`
	`$OSJI/tools.jar`
	For *Level Zero* integration, or when not using any application server, add the following:
	`$JTML/jmtlnoappserver.jar`
	For *Level One* integration add one of the following, depending on the application server you use:
	`$JMTL/jmtlwl61.jar`
	`$JMTL/jmtlwl70.jar`

Environment Variable	*Description and Setting*

`LD_LIBRARY_PATH`	Required to locate ObjectStore shared libraries on Solaris. Setting must include:
	`$OS_ROOTDIR/lib`
	`$OSJI/lib`

`LD_PRELOAD`	On HP-UX 32 bit platforms and on Solaris, you need to specify the environment variable `LD_PRELOAD` to ensure that the C++ delete operators are loaded in the correct order.
	On Solaris:
	`set LD_PRELOAD=libosopdel.so`
	On HP-UX:
	`set LD_PRELOAD=libos.sl:libosth.sl`

Building and Deploying Applications

This section describes how to build and deploy Enterprise Java Beans in supported J2EE application servers. When using Javlin, you follow the same procedure you would use with the application server alone, except that you also perform the steps outlined in this section.

Regardless of which application server you use, you must do the following:

- Code and compile your data-model classes, bean classes, and home and remote interfaces.
- Run `OSJCFP` on any classes that are to be stored in Javlin databases, making them persistence capable. Some classes are not persistence-capable but can access persistence-capable classes. Such classes should be made persistence-aware. For more information, see the *ObjectStore Java API User Guide*, Chapter 8.

After you have completed these steps, refer to the following section for server-specific procedures:

- Building and Deploying on WebLogic 6.1 or 7.0 on page 69

Note Javlin applications use Javlin's ObjectStore server. See *Managing ObjectStore* in the ObjectStore documentation set for information on starting ObjectStore servers.

Building and Deploying on WebLogic 6.1 or 7.0

If you are using the WebLogic 6.1 or 7.0 Application Server, do the following:

1 Write the beans' deployment descriptors, including the following:

- The standard EJB descriptor (`ejb-jar.xml`)
- The WebLogic-specific descriptor (`weblogic-ejb-jar.xml`)
- The Javlin-specific deployment descriptor in XML format (`jmtl-dd.xml`)

2 Generate the implementations for the home and remote interfaces by running the WebLogic `EJBC` tool on the `JAR` file created in the previous step. The output of `EJBC` is a `JAR` file that contains the deployable bean.

3 Create a manifest file to include with the `JAR` file.

4 Package the `.class` files and the deployment descriptors into a `JAR` file. The deployment descriptors must be under the `META-INF` subdirectory of the packaged `JAR` file.

5 Deploy the beans in the application server as you would for any EJB application that uses the WebLogic Server. That is, copy the bean to the `/config/`*`domain_name`*`/applications` directory of the WebLogic Administration server, where *`domain_name`* is the name of the WebLogic Server domain in which you are deploying the application.

Alternatively, you can deploy the beans with the WebLogic Administration Console.

6 Start the WebLogic server.

Chapter 7
Using the Javlin Console

The Javlin Console is used to monitor transactional activity against caches running in a Javlin application.

Contents

This chapter explains how to use the Javlin Console. It covers the following topics:

Starting the Javlin Console	71
Configuring Data Displays	75
Modifying Preferences	82
Saving and Reusing Console Settings	83
Stopping the Console	83

Starting the Javlin Console

To start the Javlin Console, the `console.jar` file must be included in your `CLASSPATH` environment variable. Define the `CLASSPATH` as follows:

On Windows:

```
set CLASSPATH=%JMTL%\console.jar;%CLASSPATH%
```

On UNIX:

```
setenv CLASSPATH $JMTL/console.jar:$CLASSPATH
```

where the `JMTL` environment variable is set to the Javlin installation directory.

To start the Javlin Console, type the following command at the Windows or UNIX command prompt:

```
java com.odi.jmtl.tools.Console
```

Alternatively, the Javlin Console can be started by using the `.jar` file as the command line argument as follows:

On Windows:

```
java -jar %JMTL%\console.jar
```

On UNIX:

```
java -jar $JMTL/console.jar
```

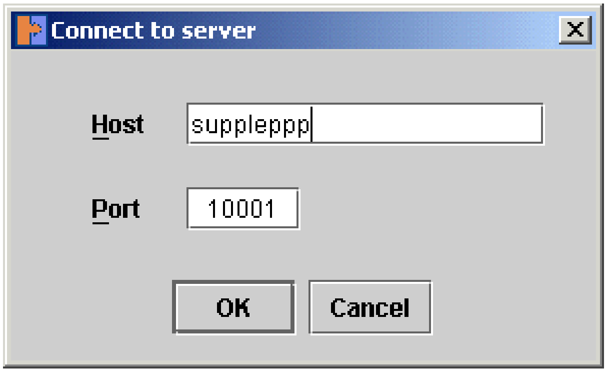
The Javlin Console is a standalone tool that does not require any of the other Javlin classes. The tool can be run from any system with a supported Java runtime environment.

Connecting to a Javlin Application

The Javlin Console communicates with your Javlin application through a socket on a specified port number. By default, every Javlin application opens port `10001` to listen for requests from the Javlin Console. If that port is already in use, the Javlin application will select another port. At startup, your Javlin application will display the port number on standard output in the following manner:

```
> java <your-application-name>
Console ServerSocket started on port: 10001
...
```

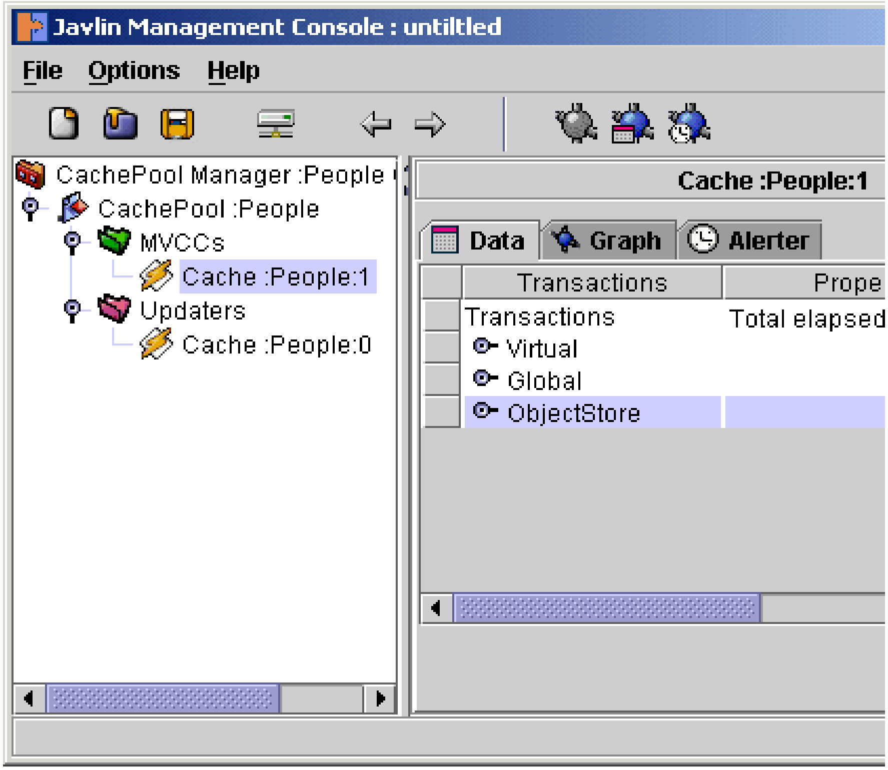
When the Javlin Console starts up, connect to your Javlin application by selecting the File | Connect menu option. This displays a dialog box with the default values for the host and port number. Type in the correct host and port number of the Javlin application you wish to monitor.



Host suppleppp

Port 10001

The Javlin Console will then display the CachePool Manager for the Javlin application on the specified host and port.



CachePool Manager :People
CachePool :People
MVCCs
Cache :People:1
Updaters
Cache :People:0

Transactions	Prope
Transactions	Total elapsed
Virtual	
Global	
ObjectStore	

The transactional activity can be summarized at multiple levels:

- CachePool Manager (all caches)
- CachePool (all caches in a CachePool)
- MVCCs (all MVCC caches in a CachePool)
- Updaters (all update caches in a CachePool)
- Cache (one cache instance)

To select a summarization level, click CachePool Manager, CachePool, MVCCs, Updaters, or Cache in the left pane.

Disabling Javlin Monitoring

You can disable the collection of monitoring information in your Javlin application by setting the `com.odi.jmtl.console` property to `false`. Add the property to the command line of your Javlin application as follows:

```
java -Dcom.odi.jmtl.console=false <your-application-name>
```

Overriding the Default Port Number

You can override the default port number when you start your Javlin application by setting the `com.odi.jmtl.console` property. Replace the *<port-number>* with the desired port number using the following command:

```
java -Dcom.odi.jmtl.console=<port-number> <application-name>
```

Type the same port number in the Javlin Console to connect to your application.

Configuring Data Displays

The Javlin Console presents the information in three different formats. You can choose to view the information on three types of panels:

- Data
- Graph
- Alerter

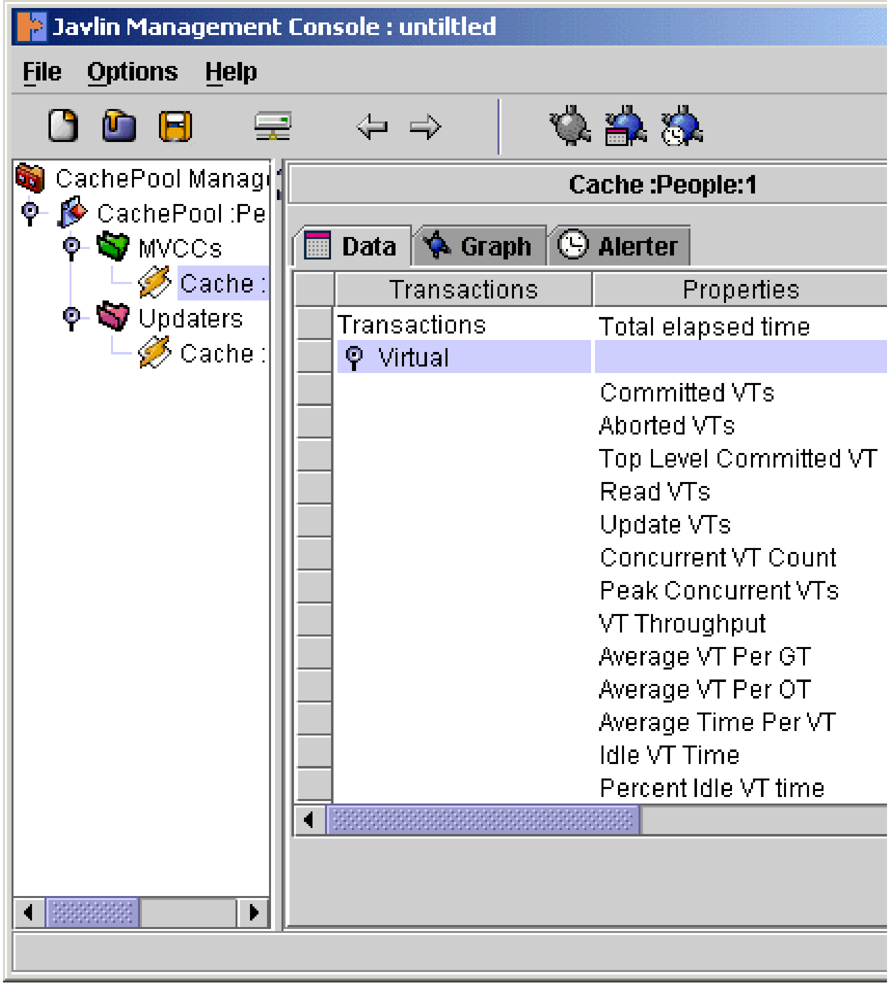
Click on the appropriate tab to select a panel. The panels and data presented are described in the following sections.

Data Panel

The Data panel displays data elements, or *transaction counters*, for three types of transactions executing against caches:

- Virtual
- Global
- ObjectStore

The dependencies between these three types of transactions are described in Global and Virtual Transactions on page 6 and Scheduling Transactions on page 58.



Transactions	Properties
Transactions	Total elapsed time
Virtual	
	Committed VTs
	Aborted VTs
	Top Level Committed VT
	Read VTs
	Update VTs
	Concurrent VT Count
	Peak Concurrent VTs
	VT Throughput
	Average VT Per GT
	Average VT Per OT
	Average Time Per VT
	Idle VT Time
	Percent Idle VT time

To view selected transaction counters on the Data panel:

1 Click on the Data tab in the right pane.

2 Click on ⊸ next to CachePool Manager, CachePool, MVCCs, Updaters or Cache in the left pane to select a summarization level.

3 Click on ⊸ next to Virtual, Global or ObjectStore in the right pane to select the transaction counters.

Virtual Transaction Counters

The virtual transaction (VT) section of the Data panel displays several data elements, or *counters*, containing information about virtual transactions that have executed or are executing in the selected cache(s). These counters can be used to analyze the performance of a Javlin application. For example, the Average VT per OT counter provides a measure of the amount of transaction batching occurring in the application.

The virtual transaction counters are described in the following table.

Virtual Transaction Counter	*Description*
Committed VTs	Number of committed virtual transactions
Aborted VTs	Number of aborted virtual transactions
Top Level Committed VTs	Number of committed virtual transactions that were not controlled by a higher-level virtual or global transaction
Read VTs	Total number of read virtual transactions
Update VTs	Total number of update virtual transactions
Concurrent VT Count	Number of virtual transactions executing concurrently
Peak Concurrent VTs	Highest number of concurrent virtual transactions
VT Throughput	Average number of virtual transactions (x1000) per second
Average VT per GT	Average number of virtual transactions controlled by a global transaction
Average VT per OT	Average number of virtual transactions scheduled together in a single ObjectStore transaction
Average Time per VT	Average time where a virtual transaction was active
Idle VT Time	Total time where no virtual transaction was active
Percent Idle VT Time	Percent of total time where no virtual transaction was active

Global Transaction Counters

The global transaction (GT) section of the Data panel displays several data elements, or *counters*, containing information about the global transactions that have executed or are executing in the selected cache(s).

Global transactions (as displayed in the Javlin Console) includes both JTA transactions and *top-level virtual transactions.* JTA transactions are initiated by a J2EE application server or bean using the `javax.jta.UserTransaction` interface, while a top-level virtual transaction is initiated by a Javlin bean using the `com.odi.jmtl.VirtualTransaction` interface. Virtual transactions are considered top-level virtual transactions when they are not controlled by a global transaction or a higher-level virtual transaction.

The global transaction counters are described in the following table.:

Global Transaction Counter	***Description***
Committed GTs	Number of all committed JTA transactions and committed top-level virtual transactions
Aborted GTs	Number of all aborted JTA transactions and aborted top-level virtual transactions
Quick Committed GTs	Number of JTA transactions and top-level virtual transactions that can be safely committed before the ObjectStore transaction is committed
Read GTs	Number of read-only JTA transactions and read-only top-level virtual transactions
Average GT Per OT	Average number of JTA transactions and top-level virtual transactions per ObjectStore transaction
Update GTs	Total number of update JTA transactions and update top-level virtual transactions
Pending GTs	Current number of pending JTA transactions and pending top-level virtual transactions

ObjectStore Transaction Counters

The ObjectStore transaction (OT) section of the Data panel contains information about the physical transaction. The ObjectStore transaction counters are described in the following table:

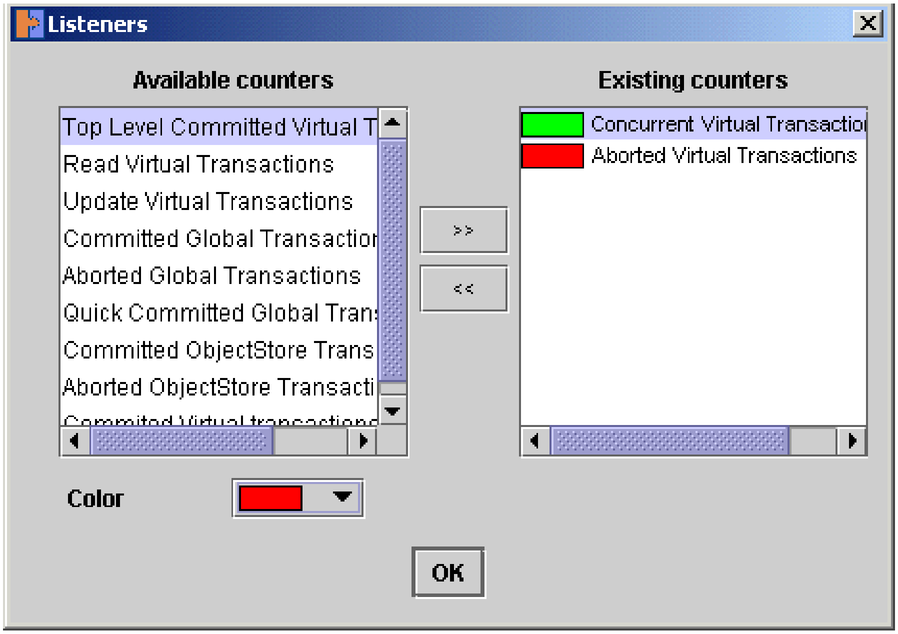
ObjectStore Transaction Counter	*Description*
Committed OTs	Number of all committed ObjectStore transactions
Aborted OTs	Number of all aborted ObjectStore transactions
Percent OT Time	Percentage of time an ObjectStore transaction is active
Checkpointed OTs	Number of all checkpoint ObjectStore transactions
OT Time	Total time of all the committed and checkpoint ObjectStore transactions
OT Throughput	Average number of ObjectStore transactions (x1000) per second

Graph Panel

The Graph panel allows you to select a particular set of data to be graphed against elapsed time. To select the data you wish to graph:

1 Click Graph to display the panel.

2 Click add + (Add/remove counter) at the bottom of the Graph panel. The Listeners dialog displays.



Top Level Committed Virtual T
Read Virtual Transactions
Update Virtual Transactions
Committed Global Transactio
Aborted Global Transactions
Quick Committed Global Tran
Committed ObjectStore Trans
Aborted ObjectStore Transacti
Commited Virtual transactions

🟩 Concurrent Virtual Transactio
🟥 Aborted Virtual Transactions

Color 🟥

Available Counters lists counters that you can select for display on the Graph panel. Existing Counters lists counters that are currently selected for display on the Graph panel.

3 To add a counter to Existing Counters, first select a color by clicking on the Color pull down menu. Next, click on a counter in the Available Counters list and then click add >> . The selected counter displays in the Existing Counters list.

4 To remove a counter from Existing Counters, select the counter from the list and then click remove << .

5 Click OK when you are satisfied with the contents of the Existing Counter list.

6 On the Graph panel, click (Start recording data) to activate the display of counters.

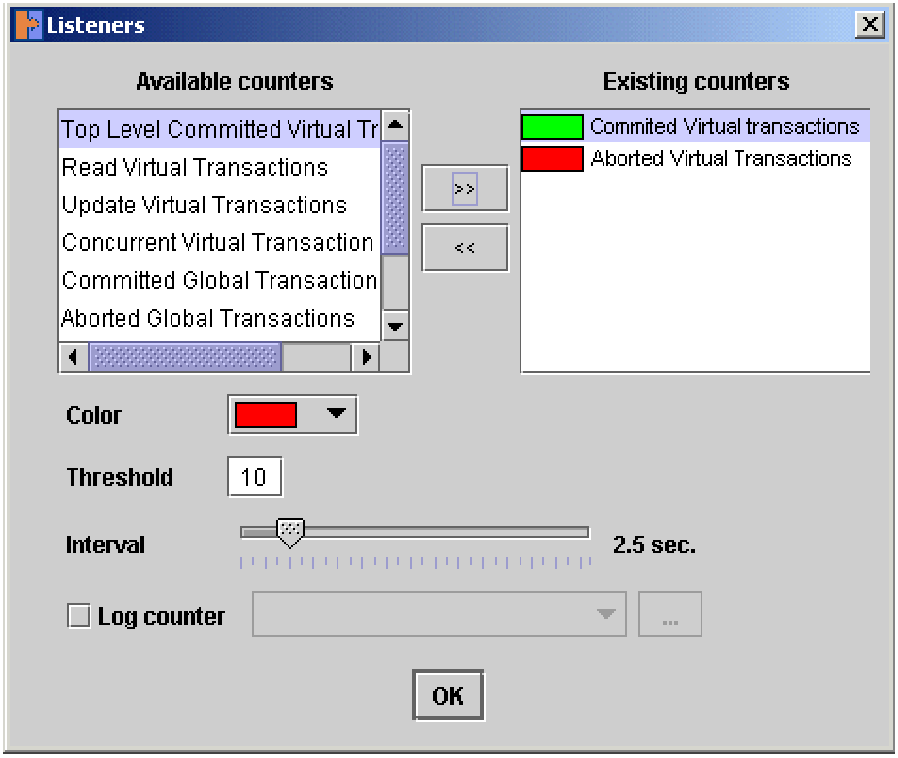
Alerter Panel

The Alerter panel allows you to select transaction counters and specify thresholds for those counters. When a counter level reaches a specified level, a warning is displayed on the Alerter panel.

To select the data you wish to be alerted on:

1 Click on Alerter to display the panel.

2 Click add (Add/remove counter) at the bottom of the Alerter panel. The Listeners dialog displays.



Listeners

Available counters

Top Level Committed Virtual Tr
Read Virtual Transactions
Update Virtual Transactions
Concurrent Virtual Transaction
Committed Global Transaction
Aborted Global Transactions

Existing counters

🟩 Commited Virtual transactions
🟥 Aborted Virtual Transactions

Color 🟥

Threshold 10

Interval 2.5 sec.

□ Log counter

Available Counters lists counters that you can select for display on the Alerter panel. Existing Counters lists counters that are currently selected for display on the Alerter panel.

3 To add a counter to Existing Counters, first select a color by clicking on the Color pull down menu. Next, type a value for Threshold and select a value for Interval. Finally, click on a counter in the Available Counters list and then click add >> . The selected counter and color display in the Existing Counters list.

4 To remove a counter from Existing Counters, select the counter from the list and then click remove << .

5 Click OK when you are satisfied with the contents of the Existing Counter list.

Logging Counters

The Log counter option allows you to log transaction counter alerts to a file. When you add a counter to the Existing Counters list, you enable logging by checking Log counter and then typing in a log file pathname before you click >> to add the counter.

Modifying Preferences

The Javlin Console allows you to modify default settings that control the frequency of data updates and the synchronization of the different display panels. Select Options | Preferences on the menu to modify these settings

Preference	*Description*
Default table update rate	Frequency of data panel update. Default is every 2.5 seconds.
Default graph update rate	Frequency of graph panel update. Default is every 2.5 seconds.
Default graph tick count	Number of intervals displayed on graph panel. Default is 30 intervals.
Default alerter/logger update rate	Frequency of alerter/logger update. Default is every 2.5 seconds.
Default alerter/logger threshold	Counter value threshold for alterer/logger. Default is 10.
Default synchronization graph rate	Frequency of graph synchronization. Default is every 2.5 seconds.
Default synchronization graph & table rate	Frequency of graph and data synchronization. Default is every 2.5 seconds.

Preference	*Description*
Default synchronization graph & alerter rate	Frequency of graph and alerter synchronization. Default is every 2.5 seconds.
Number of concurrent thread requests	Number of concurrently executed thread requests from data, graph and alterter panels. Default is 4.

Saving and Reusing Console Settings

Once you have setup display and logging settings, you can save these settings for reuse in a subsequent console session. To save console settings, go the menu bar and select File | Save. In the Save dialog, type a filename to save the console settings.

To reuse console settings in a subsequent console session, go to the menu bar and select File | Open. In the Open dialog box, type the filename for the console settings you wish to load.

Stopping the Console

To stop the Javlin Console, go to the menu bar and select File | Exit.

Index

A

access mode
  defined 7
  routing 55
  specifying 26
AddressSpaceSize attribute 18
applications, building and deploying with Javlin 68
architecture, Javlin 2
attributes
  AddressSpaceSize 18
  boolean-value 23
  byte-value 23
  CachePoolName 26
  CacheSize 18
  character-value 23
  ClassType 23
  CommitIfIdle 18
  CommitIfIdleMvcc 18
  DebugLevel 16
  DominantOperation 27
  double-value 23
  float-value 23
  GroupOpenInterval 18, 19
  integer-value 23
  IsolationLevel 26
  LockTimeout 19
  long-value 24
  MaxConcurrentTransactions 19
  MethodArguments 23
  MethodSignature (object descriptor) 23
  NumberOfMvccCaches 19, 38
  NumberOfUpdateCaches 19, 38
  object-value 24
  PhysicalName (database descriptor) 21
  PhysicalName (root descriptor) 22
  RetryLimit 27
  RootObjectDescriptor 22
  root-object-descriptor (root descriptor) 22
  short-value 24
  StorageName 22
  string-value 24
  TransactionAttribute 27
  Update 26
  UserTransactionUsage 19

B

boolean-value attribute 23
building Javlin applications 68
byte-value attribute 23

C

Cache class
  getCurrent() 35

getRootValue() 36
Cache Pool Manager
configuring 15
descriptor 15
example configuration 16
Cache Pools
configuring 17
defined 5
descriptor 17
example configuration 19
monitoring 71
naming with method descriptor 26
NumberOfMvccCaches attribute 38
NumberOfUpdateCaches attribute 38
Cache Storage
configuring 20
example configuration 24
CachePool attribute 38
CachePool class
createDatabase() 35
CachePoolConfiguration class 34
CachePoolManager class 34
getCachePool() 35
setDBMapping() 35
CachePoolManagerConfiguration class 34
CachePoolName attribute 26
caches
defined 5
monitoring 71
multicache configuration 38
MVCC, defined 5
MVCC, working with 37
routing 55
single-cache configuration 34
update 5
CacheSize attribute 18
character-value attribute 23
CLASSPATH
level one integration 65
level zero integration 64
UNIX 67
Windows 66
ClassType attribute 23
CommitIfIdle attribute 18
CommitIfIdleMvcc attribute 18, 60
committing virtual transactions 42
component methods, configuring 26
component-descriptor tag 25
configuration file
deployment-descriptor names 14
external entities 13
validating 12
XML format 12
configuring
Cache Pool 17
Cache Pool Manager 15
Cache Storage 20
component methods 26
deployed components 25
Javlin 11
loading at run time 11
console
configuring displays 75
defined 71
preferences 82
saving settings 83
starting 71
stopping 83

D

database roots
creating 35
descriptor 22, 36
logical name 34
retrieving 35
databases
creating 35
descriptor 21
logical name 34
path change 35
DebugLevel attribute 16

default method descriptor 26
deployed components
  configuring 25
  example configuration 27
deploying Javlin applications 68
descriptors
  Cache Pool 17
  Cache Pool Manager 15
  database 21
  default method 26
  loading at run time 11
  method 26
  object 22
  root 22, 36
DOCTYPE statement 12
dominant transaction 7
DominantOperation attribute 27
double-value attribute 23
DTD location 12

E

entity reference 13
ENTITY statement 13
environment variables
  CLASSPATH 64, 65
  JMTL 62
  OS_ROOTDIR 62
  OSJI 62
  shared library path 62
  UNIX 67
  Windows 66
example deployment descriptors
  Cache Pool 19
  Cache Pool Manager 16
  Cache Storage 24
  deployed components 27
  Javlin configuration 29
external entities 13

F

float-value attribute 23

G

global transactions
  defined 6
GlobalTransaction class
  AbortAndRetryException 43
  RetryException 43
GroupOpenInterval attribute 18, 19
GroupOpenIntervalMvcc attribute 59

I

initializing configuration information 11
integer-value attribute 23
isolation levels
  defined 8
  listed 8
  routing 55
  scheduling 58
  specifying 26
IsolationLevel attribute 26

J

Javlin
  basic tasks 4
  building and deploying applications 68
  declarative configuration 11
  example configuration 29
  overview 1, 71
  process architecture 2
  programmatic configuration 33

L

level one integration
  CLASSPATH 65
  defined 3
level two integration

defined 4

level zero integration

CLASSPATH 64

defined 3

levels, isolation

defined 8

listed 8

routing 55

scheduling 58

specifying 26

loading configuration information 11

local transactions 6

location of DTD 12

`LockTimeout` attribute 19

logical database name 34

logical root name 34

`long-value` attribute 24

M

`MaxConcurrentTransactions` attribute 19

method descriptor 26

`MethodArguments` attribute 23

`MethodSignature` attribute

object descriptor 23

monitoring

caches 71

disabling 74

MVCC caches

defined 5

routing 56

routing restrictions 38

using 37

N

`noUserTransactions` value 19

`NumberOfMvccCaches` attribute 19, 38

`NumberOfUpdateCaches` attribute 19, 38

O

object descriptor 22

`object-value` attribute 24

Operation Context

creating 41

optimizing transactions 49

`OSJCFP` 68

P

performance

improving 49

monitoring 71

persistence

creating databases 35

making classes persistence capable 68

`PhysicalName` attribute

database descriptor 21

root descriptor 22

post-processing 68

process architecture, Javlin 2

programming restrictions

accessing persistent data 48

retaining persistent references 47

thread synchronization 49

use of OSJI classes 47

R

retry limit, setting 27

`RetryLimit` attribute 27

`RootMapping` 34

`RootObjectDescriptor` attribute 22

`root-object-descriptor` attribute

root descriptor 22

roots, database

creating 35

descriptor 22, 36

logical name 34

retrieving 35

routing

caches 55
defined 8
isolation levels 57
restrictions 38

S

scheduling 58
CommitIfIdleMvcc 60
GroupOpenIntervalMvcc 59
transactions 58
virtual transaction concurrency 58
short-value attribute 24
specifying DTD location 12
StorageName attribute 22
string-value attribute 24

T

tasks, basic Javlin 4
threads
synchronization restrictions 49
transactional caching 4
TransactionAttribute attribute 27
transactions
aborting 42
attribute, setting 27
beginning 41
committing 42
dominant 7
example 43
global 6
local 6
optimizing 49
retrying 43
virtual 6
TransactionTry 44

U

Update attribute 26
update caches 5
userTransactionsPossible value 19
UserTransactionUsage attribute 19

V

validating configuration file 12
virtual transactions
aborting 42
accessing persistent data 48
beginning 41
committing 42
concurrency 58
exception handling 43
retaining persistent references 47
retrying 43
use of OSJI classes 47
VirtualTransaction class
abort() 42
begin() 41
commit() 42
isActive() 43
RetryException 43

X

XML statements
DOCTYPE 12
ENTITY 13