.NET Developer's Reference

VERSION 2020.0.02

Contents

Before You Begin

The *JADE NET Developer’s Reference* is intended as a major source of information when using the JADE .NET class library to develop .NET applications accessing JADE database objects.

Who Should Read this Reference

The main audience for the *JADE .NET Developer’s Reference* is expected to be .NET developers using the .NET Framework.

What’s Included in this Reference

The *JADE .NET Developer’s Reference* has seven chapters and two appendices.

Chapter 1	The software requirements for developing and running JADE .NET applications
Chapter 2	A reference to the JADE Object Manager functionality available from JADE .NET
Chapter 3	An introductory tutorial to JADE .NET
Chapter 4	An overview of the .NET example for the Erewhon Demonstration System
Chapter 5	An overview of the framework for developing .NET applications to connect to a JADE system
Chapter 6	A reference to the .NET classes resulting from exposing JADE classes
Chapter 7	A reference to the **JadeSoftware.Jade.DotNetInterop** library and the API required to use the exposed classes
Appendix A	A mapping of JADE primitive types to Common Language Runtime (CLR) data types
Appendix B	A reference to developing applications to store, edit, and query spatial information directly through the JADE .NET API

Related Documentation

Other documents that are referred to in this reference, or that may be helpful, are listed in the following table, with an indication of the JADE operation or tasks to which they relate.

Title	Related to…
JADE Database Administration Guide	Administering JADE databases
JADE Developer’s Reference	Developing or maintaining JADE applications
JADE Development Environment Administration Guide	Administering JADE development environments
JADE Development Environment User’s Guide	Using the JADE development environment
JADE Encyclopaedia of Classes	System classes (Volumes 1 and 2), Window classes (Volume 3)
JADE Encyclopaedia of Primitive Types	Primitive types and global constants
JADE Installation and Configuration Guide	Installing and configuring JADE

Title	Related to…
JADE Initialization File Reference	Maintaining JADE initialization file parameter values
JADE Object Manager Guide	JADE Object Manager administration
JADE Synchronized Database Service (SDS) Administration Guide	Administering JADE Synchronized Database Services (SDS), including Relational Population Services (RPS)
JADE Thin Client Guide	Administering JADE thin client environments
JADE Web Application Guide	Implementing, monitoring, and configuring Web applications

Conventions

The *JADE .NET Developer's Reference* uses consistent typographic conventions throughout.

Convention	Description
Arrow bullet (»)	Step-by-step procedures. You can complete procedural instructions by using either the mouse or the keyboard.
Bold	Items that must be typed exactly as shown. For example, if instructed to type **foreach**, type all the bold characters exactly as they are printed. File, class, primitive type, method, and property names, menu commands, and dialog controls are also shown in bold type, as well as literal values stored, tested for, and sent by JADE instructions.
Italic	Parameter values or placeholders for information that must be provided; for example, if instructed to enter *class-name,* type the actual name of the class instead of the word or words shown in italic type. Italic type also signals a new term. An explanation accompanies the italicized type. Document titles and status and error messages are also shown in italic type.
Blue text	Enables you to click anywhere on the cross-reference text (the cursor symbol changes from an open hand to a hand with the index finger extended) to take you straight to that topic. For example, click on the "How a .NET Application Connects to JADE" cross-reference to display that topic.
Bracket symbols ([ ])	Indicate optional items.
Vertical bar (|)	Separates alternative items.
`Monospaced font`	Syntax, code examples, and error and status message text.
ALL CAPITALS	Directory names, commands, and acronyms.
Small font	Keyboard shortcut keys.

Key combinations and key sequences appear as follows.

Convention	Description
Key1+Key2	Press and hold down the first key and then press the second key. For example, "press Shift+F2" means to press and hold down the Shift key and press the F2 key. Then release both keys.

Convention	Description
Key1,Key2	Press and release the first key, then press and release the second key. For example, "press Alt+F,X" means to hold down the Alt key, press the F key, and then release both keys before pressing and releasing the X key.

Chapter 1 Software Requirements

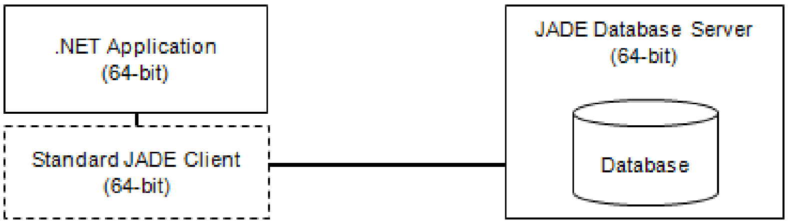
This chapter covers the following topics.

- JADE Requirements
- .NET Requirements

JADE Requirements

When a .NET application uses classes that have been exposed from a JADE system, it uses the APIs from the **JadeSoftware.Jade.DotNetInterop.dll** and other assemblies in the JADE .NET framework for JADE version 7.0.03 or later. These assemblies are 64-bit DLLs.

The .NET application connects to the JADE database using the binary files from a standard JADE client, which must be a 64-bit client.



Note The C# exposure can be generated using any JADE development client (32-bit or 64-bit).

.NET Requirements

To develop and compile .NET applications for .NET exposure, you require a *minimum* of:

- Microsoft Visual Studio 2017 or higher

Note Projects must target .NET Framework 4.7.1 or higher.

For external component libraries, you require:

- .NET 4.x components (or .NET 3.x, if you are using components designed for .NET 3.x)

Chapter 2 Object Management

This chapter covers the following topics.

Overview

This chapter describes the JADE Object Manager, which implements object-oriented and distributed processing functionality, including features such as object locking, transactions, transaction isolation, and automatic cache coherency concurrency.

In this chapter, the term *JADE session* or *JADE session execution* is used to refer to a *JADE process* in a node. This terminology should assist the .NET developer who is unfamiliar with the JADE and could potentially confuse a JADE process with a Windows process.

JADE Object Handling

The JADE distributed processing architecture has a single semantic model for all objects and two basic lifetime variations: persistent and transient lifetimes.

Persistent objects, which are permanently stored in the JADE database, live across JADE session executions and are shared by all JADE sessions in all nodes of the system. Concurrency control is enforced for persistent objects. Update operations on persistent objects must be performed within a transaction; typically inside a **JoobContext BeginTransaction** and **System.Data.IDbTransaction Commit** method pair.

Changes made to persistent objects within a transaction are hidden from other JADE sessions until committed. This means that other JADE sessions accessing these objects will not see the uncommitted updates, but will instead view the most-recently committed editions.

Transient objects are local to the JADE session that created them; that is, they cannot be accessed by any other JADE session. Because of this, no concurrency control operations are performed when they are updated, which gives optimal performance. These transient objects are automatically removed when the JADE session that created them becomes inactive, typically by disposing its attached **JoobContext** instance.

No transaction is necessary when creating, deleting, or updating transient objects. These objects can be updated at any time.

By default, JADE objects are created with persistent lifetime. To create transient objects, there is a constructor for a JADE object that takes the lifetime of the object as a parameter using the **ClassPersistence** enumeration, which has the values **Persistent** and **Transient**.

JADE Object Caches

On each JADE client node, there is a persistent JADE object cache to hold in-memory copies of persistent objects that have been accessed by JADE sessions, through their associated JADE context, on that node. One of the benefits of caching an object is that the object can be accessed via JADE contexts attached to the client node at a later time without needing to be fetched from the database server node.

Because objects are retained in cache, it is possible to run out of allocated cache space. When this happens, the least-recently used objects are discarded from cache. If a discarded object is required in the future, the latest edition is fetched from the database.

JADE automatic cache coherency keeps objects in the cache from becoming out of date. When a JADE object is updated, the database server knows which connected caches have copies of the object that are now obsolete. It sends messages to the affected caches, resulting in obsolete objects being discarded. If a discarded object is required in the future, the latest edition is fetched from the database server.

Locking an object guarantees that the copy in cache will be the latest edition.

JADE also maintains a transient JADE object cache on each client node, to contain transient objects created by JADE sessions on that node. Even though cached, the transient objects are visible only to the JADE sessions that created them.

If the transient JADE object cache of a client node overflows, transient objects are written out to a **.tmp** file. Each transient database file is unique to a specific JADE session and is located in the path specified by the **TransientDbPath** parameter in the [JadeClient] section of the JADE initialization file.

The file name has the following format.

```
tdb_<host-name>_<pid><designator>.tmp
```

In this format, the ***<pid>*** value is the operating system executing process identifier and the ***<designator>*** value is one of the following.

- [kbp_*nnnnnn*], for transient files associated with a JADE background session
- [*session-oid*], for transient files associated with a JADE session, where *session-oid* is the object identifier of the JADE session

If a **.tmp** file is found to be growing, you can identify the JADE session to which it belongs by locating the session identifier in the **jommsg.log**; for example:

```
2014/08/25 15:58:14.399 0b958-bc0c Jom: Local process sign-on: oid=[187.251],
process=0x000000000D39DDD0, no=2, id=48140, db=4, type=2, (non-prodn),
scm=BankingViewSchema, app=DotNetConnection
```

In this log entry example, **[187.251]** is the object identifier of the JADE session. Note that in the message logs, JADE sessions are referred to as *local processes*.

The temporary (**.tmp**) file for each session is deleted when the client node terminates. Certain kinds of failure (for example, a node crash) can result in these temporary files not being deleted. You can delete any temporary transient object overflow file that has been left behind, if required.

You can set the size of the JADE caches in the JADE initialization file used by the client node.

- The default size for the persistent object cache (that is, the default value of the **ObjectCacheSizeLimit** parameter in the [JadeClient] and [JadeServer] sections of the JADE initialization file) is 80M bytes, and 40M bytes for the transient object cache (that is, the default value of the **TransientCacheSizeLimit** parameter in the [JadeClient] and [JadeServer] sections of the JADE initialization file).
- The minimum size for the persistent and transient object caches is 3M bytes.

JADE Object Concurrency Management

When multiple threads are accessing and updating JADE database objects at the same time, there are concurrency issues that need to be managed, as follows.

- Data must be kept consistent when being updated.
- Applications need to access consistent, up-to-date data.

JADE provides features such as object locking, transactions, transaction isolation, and automatic cache coherency to address these issues.

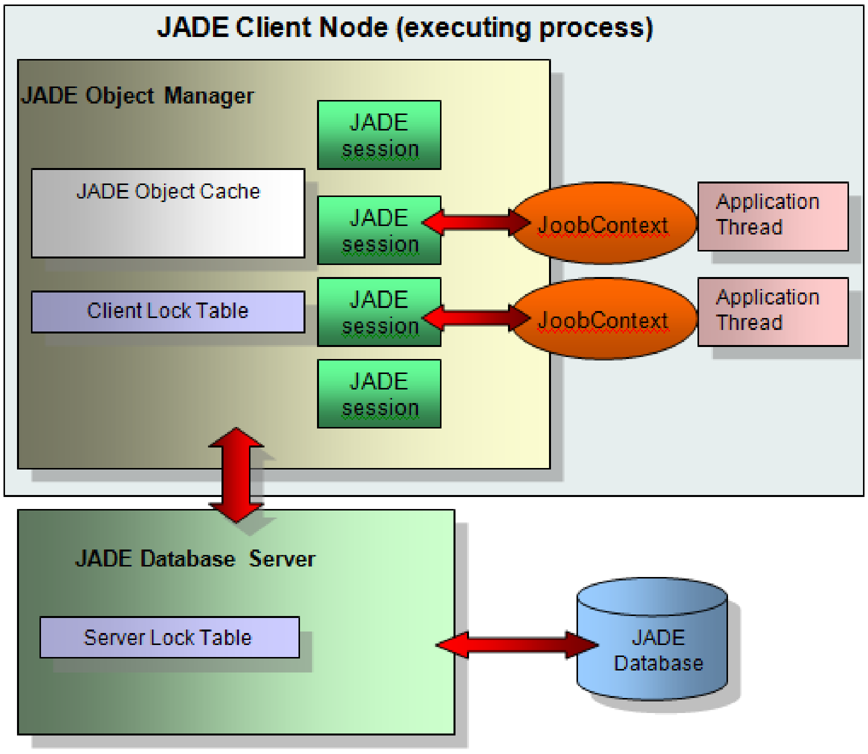
Many of the features work automatically, but there are various options to make them more efficient.

This chapter discusses these features and options, describing how to use this area of functionality and the options that are available to write efficient and scalable applications. Topics include:

- Distributed processing architecture
- Maintaining data consistency and currency
- Automatic cache coherency
- The transaction model
- Transaction isolation
- Object locking
- Object volatility (stable and frozen objects)
- Frozen files and partitions

Distributed Processing Architecture

The following diagram is an overview of the JADE distributed processing architecture.



For the purposes of this chapter, a JADE system consists of a JADE database, a JADE database server, and one or more JADE client nodes. Each client node is an executing process containing a pool of JADE sessions that allow threads to connect to JADE using **JoobContext** instances. There is also a cache to hold local copies of JADE objects accessed by JADE sessions on that client node, and a lock table to record which objects are currently locked by active sessions on that node.

The JADE database server has a master lock table to record JADE object locks held by applications running on all nodes.

Nodes

A JADE node is an executing process. There are two main types of nodes: the *server* node and *client* node.

The server node has a direct connection to the JADE database and maintains system-wide information (for example, which objects are currently locked). Client nodes connect to the server node for JADE object locking, retrieval, and storage. An executing process that connects to the JADE database server is a client node.

JoobContexts, Sessions, and JoobConnections

Each client node has a pool of JADE sessions, providing an interface to the JADE Object Manager and JADE database.

A process thread attaches to an available JADE session by creating a **JoobContext** object. The thread is detached from the JADE session when the **JoobContext** object is disposed.

A **JoobContext** object interacts with a JADE session using a **JoobConnection** object. **JoobConnection** objects are typically created and disposed of automatically with their owning **JoobContext** object. In many scenarios, users will not have to explicitly interact with **JoobConnection** objects.

The C# **JoobSession**, **JoobNode**, and **JoobSystem** classes are the wrappers for the JADE Object Manager **Process**, **Node**, and **System** classes, respectively. When a **JoobContext** or a **JoobConnection** class is created for first time, the JADE Object Manager initialize process occurs and a pool of JADE Object Manager process agents is created (the default value is **10**).

When your code gets a new **JoobContext** (depending on overload), a new **JoobConnection** is created, which obtains a JADE Object Manager process agent from the pool.

The **JoobContext** is the primary way of interacting with a process agent; for example, for retrieval of a specific instance, locking, notifications, and so on.

When the **JoobContext** is disposed of (explicitly or by a using block), the JADE Object Manager process agent is put back in the pool. A **JoobContext** provides access to the **JoobSession** object, if needed. It can be obtained with **JoobContext.GetSystemVariables().Process** to make the ObjectId available, which **JoobContext.FindInstance<JoobSession>(ObjectId);** can then retrieve. The grace period is the amount of time given to the process running on the node to complete and dispose of any **JoobContexts** before the node is terminated and the JADE Object Manager starts terminating the process agents. (The **TimeSpan** wait time is usually in the range 5 through 30 seconds, depending on the size and structure of your system.) For an example, see "Locking", in Chapter 7.

The C# **JoobObject** class (and all of its subclasses) has an **ObjectId** property, which is equivalent to the JADE Object Manager objects oid.

JADE Object Caches

Each client node has a JADE object cache to hold in-memory copies of JADE database objects.

When an object is retrieved from the JADE database, it is stored locally in the client node's JADE object cache. The object can be subsequently accessed locally, without needing to be retrieved via the database server.

The JADE object cache is shared by all JADE sessions for that client node.

An object in the JADE object cache could become out of date if the object is updated on another client node. However, *automatic cache coherency* and *object locking* keep the JADE object cache up to date, as described later in this chapter.

Object Locking

Object locking is used to coordinate accessing and updating of JADE objects. There are four types of locks: two for *reading* (**Shared** and **Reserve**) and two for *updating* (**Update** and **Exclusive**). Locking is explained in more detail later in this chapter.

The current lock status of JADE objects is system-wide, and is maintained in tables on the server and client nodes.

The server lock table records the locks for all JADE objects, regardless of the client node on which the locking JADE session is executing. Client nodes therefore need to communicate with the server node in order to lock and unlock JADE objects.

Each client node lock table records copies of the JADE objects that are locked by active JADE sessions on the node.

Maintaining Data Consistency and Coherence

This section describes data consistency and data coherence.

Data Consistency

When a JADE session is updating JADE objects, the data must be kept consistent. JADE sessions wanting to update the same object must not interfere with each other, and other sessions viewing the data must not see incomplete updates.

Object locking is used to ensure that only one JADE session at a time can be updating a JADE object. A JADE session must lock an object before it can update it, with a lock type that is available only to one JADE session at a time (that is, an *exclusive* or an *update* lock).

JADE sessions can update objects only within a transaction; that is, while the transaction is active (for example, bracketed by **JoobContext BeginTransaction** and **IDbTransaction Commit** method calls).

While a transaction is active, any updates are not visible to other JADE sessions. The other JADE sessions instead view the JADE objects, as they were without the uncommitted updates. Committing the transaction causes the updates to be applied to the JADE database and made visible. This is discussed in more detail later in this chapter.

Data Coherence

When a JADE session accesses a JADE object that has been copied into the JADE object cache on a client node, the copy may no longer be current if the object has been updated on another client node. The object will need to be fetched again from the database server node, to bring it up to date.

Automatic cache coherency (discussed in the following section) is a JADE feature that keeps objects in the JADE object cache up to date.

Object locking can also be used to ensure that a JADE object is up to date. When a JADE session has an object locked, its view of the object is guaranteed to be completely up to date.

Automatic Cache Coherency

JADE objects being used by JADE sessions are copied into local memory, in the JADE object cache of the client node. This speeds up access, by minimizing object requests to the database server node. However, this creates the situation where if the object is updated on another client node, the local copy in cache could no be longer up to date.

To handle the situation where the local copy in cache would otherwise become out of date, JADE provides a feature, *automatic cache coherency*, which automatically detects when a JADE object has been updated and causes local copies of the object to be marked for refresh in all of the caches where it has been copied. This means that the next time the object is accessed, it is fetched from the database server node and therefore be up to date. This feature is enabled by default.

For most read operations, this mode of operation is satisfactory and avoids the extra code and overhead involved with having to place object locks. However, there is a small time delay between when the object is updated in the database and when the client nodes are notified. If it is crucial to an application that a JADE object being read is guaranteed to be up to date, a JADE session must lock the object.

Note Automatic cache coherency does not apply to JADE collections, which are kept up to date by being automatically locked whenever accessed.

The Transaction Model

The transaction model is used to group and isolate sets of updates to database objects, in order to maintain data consistency.

All sets of updates must be applied and made visible as a group. Until the updates are ready, they should be invisible to other JADE sessions. If something goes wrong, all of the pending updates must be discarded.

In order to update objects, a JADE session must begin a transaction. The updates are held temporarily until the transaction is committed, at which time they get applied to the JADE database as permanent changes and made visible. Before that happens, other JADE sessions view the objects as they were without the uncommitted updates (this is termed *transaction isolation*).

If the full set of updates cannot be completed (for example, due to an error), the transaction is rolled back, which cancels the transaction and discards all of the pending updates in the set.

This concept implements the standard transaction ACID principles of atomicity (either all of the updates or none of the updates get applied), and isolation (the updates are not visible until applied).

With JADE, a transaction is typically started using the **JoobContext BeginTransaction** method. This method returns a **System**.**Data**.**IDbTransaction** JADE transaction object. The transaction object's **Commit** method commits the transaction, which applies the updates to the database. Alternatively, the **Rollback** method cancels the transaction, discarding the uncommitted updates.

The transaction object must be disposed of at the end of the transaction. If **Commit** is not called before the transaction object is disposed of, the transaction will be rolled back.

You can use a **using** block to carry out a transaction and ensure that the transaction object is disposed of at the end of the block.

The following code fragment illustrates a successful transaction, where updates are permanently applied to the JADE database and made visible.

```
using (JoobContext context = new JoobContext())
{
    using (System.Data.IDbTransaction tran = context.BeginTransaction())
```

```
        {
            //... update objects
            tran.Commit();
        }
    }
```

The following code fragment illustrates a transaction that is rolled back, where updates are discarded and not applied.

```
using (System.Data.IDbTransaction tran = context.BeginTransaction())
{
    //... update objects
    tran.Rollback();
}
```

Transaction Isolation

Transaction isolation is the mechanism that makes updates that have not yet been committed invisible to other JADE sessions. Instead, these JADE sessions view the most-recently committed editions of the JADE objects; that is, the way they were without the uncommitted updates.

While a JADE object is being updated, two copies can exist in the JADE object cache: one with and one without the updates. The copy with the updates is visible only to the updating JADE session. Other JADE sessions access the other copy, which does not include the updates.

When a transaction is committed, any older copies of objects in the cache are removed.

JADE automatically provides transaction isolation for JADE objects.

Object Locking Overview

JADE objects are locked individually.

Locking a JADE object does two things. Firstly, it controls concurrent access to an object. Secondly, it ensures that a JADE session accesses the most up-to-date edition of the object.

JADE provides four types of lock.

- Shared
- Reserve
- Update
- Exclusive

Update and exclusive locks are *write* locks, which allow JADE sessions to update an object and prevent other sessions from updating the object. An exclusive lock also prevents the acquisition of *read* locks.

Shared and reserve locks are *read* locks, which prevent other JADE sessions from applying updates to an object without prohibiting other *read* locks.

Lock types are discussed in more detail in the following section.

Although JADE sessions can access a JADE object without locking, they cannot update it, and the view of the object's data can change if it is updated by another session.

If a JADE session does not want to update an object but wants to have a consistent view that will not change while it is being accessed, it must obtain a lock on the JADE object (of any type). The object cannot have updates committed by another JADE session until the object is unlocked.

If a JADE session wants to update a JADE object, this must be done within a transaction and an exclusive or an update lock must first be acquired, either implicitly or explicitly. This ensures that no other JADE session can update the object, as only one session at a time is allowed an exclusive or update lock on an object. In addition, the lock always remains in place until the transaction has been committed or rolled back.

JADE automatically acquires an implicit exclusive or update lock for any object being updated, if it is not already so locked.

When a JADE session requests a type of lock that is not compatible with the current locks in place by other sessions, it is queued until those locks get unlocked (up to a specific time limit) and the session can acquire the requested lock.

The locking concept helps to implement the standard transaction ACID principle of consistency (that is, data is always consistent).

JADE Object Locking

This section describes and discusses object locking in detail. Object locking applies to JADE objects.

Lock Types

There are four lock types, each providing different levels of concurrency and updating ability. For details, see the following subsections.

Shared Locks

A shared lock allows several JADE sessions to simultaneously read a JADE object but not update it.

Shared locks maintain concurrency while ensuring that a JADE session never works with obsolete data.

Reserve Locks

A reserve lock is available for situations where a JADE session eventually intends to update a JADE object but needs to minimize the length of time the object is locked with an exclusive or update lock.

Using a reserve lock allows other JADE sessions to acquire shared locks, but not any other type of lock.

Reserve locks also provide a way to avoid potential deadlocks (for details, see "Avoiding Deadlock Exceptions", later in this chapter).

Update Locks

An update lock allows a JADE session to update a JADE object, while still allowing other sessions to acquire shared locks (but not any other type of lock). The other JADE sessions with shared locks see the most-recently committed edition of the object, without the pending updates.

Update locks can be used only within a transaction.

When a JADE session commits a transaction, any update locks are automatically requested to be upgraded to exclusive locks before the updates are committed. This ensures that no other JADE sessions have the objects locked at the time the updates are committed.

Exclusive Locks

An exclusive lock allows a JADE session to update a JADE object, while preventing all other sessions from locking the object.

Lock Type Compatibility

The valid concurrent lock combinations are shown in the following table.

	Exclusive	**Update**	**Reserve**	**Shared**
Exclusive	No	No	No	No
Update	No	No	No	Yes
Reserve	No	No	No	Yes
Shared	No	Yes	Yes	Yes

To summarize:

- A JADE object can have multiple shared locks, but only one reserve, update, or exclusive lock.
- Shared locks are compatible with reserve locks and update locks, but not with exclusive locks.
- A reserve lock is compatible with shared locks, but not with update locks or exclusive locks.
- An exclusive lock is not compatible with any other lock type.

Lock Duration

A lock can have *transaction* duration or *session* duration.

A *transaction* duration lock will be unlocked when a transaction ends (that is, it is committed or rolled back). The lock will not be unlocked at the end of a transaction if it has *session* duration.

Locks with *session* duration remain in place until unlocked explicitly, or when the JADE session becomes inactive, which typically happens when its JADE context is disposed of. An explicit unlock request will unlock an object regardless of the duration, unless the JADE session currently has a transaction in effect, in which case the unlock request is ignored.

Explicit locks specify the lock duration. Implicitly acquired locks have *transaction* duration.

Lock Timeout

When a JADE session requests a lock on a JADE object that is currently incompatibly locked by another session, the request is queued until no incompatible locks remain, in which case the request can be granted. However, the lock request is queued only for a specific amount of time before it is rejected and a **JoobObjectLockedException** is thrown.

The timeout value for lock requests is specified in a parameter of the **JoobContext Lock** method. For locks that are implicitly acquired by JADE, the timeout value is specified in the following section and parameter of the JADE initialization file.

```
[JadeServer]
ServerTimeout=nnnnn
```

The **ServerTimeout** parameter specifies the maximum number of milliseconds to wait before an implicit lock request times out. The default value is 10,000 milliseconds.

Lock Kind

As well as duration and type, locks have a *kind*. Normally, the kind is zero (**0**), but can have a different value when stable objects are being used. JADE sessions have no direct control of the lock *kind*. It is of interest only when analyzing current locks.

The possible values are listed in the following table.

Kind	Description
LockKind.Normal	Normal lock
LockKind.Local	Local lock – a shared lock on a stable JADE object for a specific JADE session
LockKind.Node	Node lock – a shared lock on a stable JADE object, recorded for a client node rather than a specific JADE session

For more details, see "Stable Objects" and "Object Volatility", later in this chapter.

Explicit Locking and Unlocking

The **Lock** method of the **JoobContext** class is used to explicitly lock objects.

```
void Lock(JoobObject   receiver,
          LockType     type,
          LockDuration duration,
          TimeSpan     waitTime);
```

LockType and **LockDuration** are enums with the following sets of values.

- LockType.Shared
- LockType.Reserve
- LockType.Update
- LockType.Exclusive
- LockDuration.Transaction
- LockDuration.Session

Objects are explicitly unlocked using the **JoobContext Unlock** method.

```
void Unlock(JoobObject receiver);
```

The **Unlock** method call is ignored if the JADE session is in transaction state (that is, it has a JADE transaction in effect) or in load state (that is, it is executing within a **BeginLoad**/**EndLoad** bracket, described under "Load State"). Instead, provided that the lock is of transaction duration, the object will be unlocked when the transaction is committed or rolled back, or when load state is terminated.

Session duration locks must be unlocked using the **Unlock** method outside transaction state.

Note All locks held by a JADE session are released, regardless of duration, when the session becomes inactive, which typically is when its **JoobContext** instance is disposed of.

The following code fragment locks two JADE objects, then unlocks them:

```
//Request a shared lock, transaction duration, 1 second time out.
context.Lock(obj1, LockType.Shared, LockDuration.Transaction,
             TimeSpan.FromSeconds(1));

//Request a reserve lock, session duration, 5 second time out.
context.Lock(obj2, LockType.Reserve, LockDuration.Session,
             TimeSpan.FromSeconds(5));

context.Unlock(obj1);
context.Unlock(obj2);
```

Implicit Locking and Unlocking

This section describes implicit locking and unlocking.

Implicit Locking

JADE acquires locks implicitly in certain situations.

Whenever a JADE collection is accessed, a shared lock is implicitly acquired on the collection while it is being accessed, if it is not already locked. The implicit shared lock is removed when the access is complete, unless the JADE session is in transaction state or in load state.

When a JADE session is in transaction state, an implicit exclusive or update lock is acquired for any JADE object before it is updated, if it is not yet locked with one of these two types. By default, the implicit lock type is exclusive. However, you can use the **JoobContext SetImplicitUpdatingLockType** method to specify that update locks should be the default for objects that are implicitly locked; for example:

```
context.SetImplicitUpdatingLockType(LockType.Update);
```

The implicit lock type reverts to the default value (exclusive) when the JADE session becomes inactive (that is, when the JADE session is relinquished back to the pool of available sessions).

Implicit lock requests have transaction duration and they use the default timeout value (as specified in the **ServerTimeout** parameter in the [JadeServer] section of the JADE initialization file).

Implicit Unlocking

When a JADE session terminates a transaction (that is, it is committed or rolled back), all of its transaction duration locks are released. This happens regardless of whether the lock was acquired outside of or within transaction state, and whether or not the **Unlock** method was used within the transaction.

Session duration locks are not released when a transaction ends. You must explicitly unlock them, using the **Unlock** method.

Ending load state also implicitly unlocks transaction duration locks that were acquired while in load state.

When a JADE session becomes inactive (typically when its **JoobContext** instance is disposed of), all objects that it still has locked are unlocked, regardless of duration.

Examples

The following code sample demonstrates explicit and implicit locking, within and outside of transaction state.

```
// Lock objects outside transaction state.
TimeSpan timeOut = TimeSpan.FromSeconds(10);
context.Lock(obj1, LockType.Reserve, LockDuration.Transaction, timeOut);
context.Lock(obj2, LockType.Shared, LockDuration.Transaction, timeOut);
context.Lock(obj3, LockType.Shared, LockDuration.Session, timeOut);

using (System.Data.IDbTransaction tran = context.BeginTransaction())
{
    // Lock an object in transaction state.
    context.Lock(obj4, LockType.Exclusive, LockDuration.Transaction, timeOut);

    coll1.Add(obj1); // Acquires an implicit exclusive lock on coll1.

    // Acquire an explicit update lock. This can only be done in transaction state.
    context.Lock(coll2, LockType.Update, LockDuration.Transaction, timeOut);

    coll2.Add(obj2); // No implicit lock, as coll2 is already locked.

    context.Unlock(obj2); // This is ignored in transaction state.
    context.Unlock(obj3); // This is ignored in transaction state.

    obj5.Attr1 = "X"; // Acquires an implicit exclusive lock.

    tran.Commit(); // Upgrades coll2’s update lock to exclusive,
                   // commits the updates,
                   // then unlocks coll1, coll2, obj1, obj2, obj4, and obj5.
                   // obj3 remains locked as it has session duration.
}
context.Unlock(obj3); // This will unlock obj3.

// Specify implicit update locks are to be used.
context.SetImplicitUpdatingLockType(LockType.Update);
using (System.Data.IDbTransaction tran = context.BeginTransaction())
{
    coll3.Add(obj2); // Acquires an implicit update lock on coll3.

    tran.Commit(); // Upgrades the update lock to an exclusive lock,
                   // commits the update,
                   // then unlocks coll3.
}
context.SetImplicitUpdatingLockType(LockType.Exclusive);
```

Load State

A JADE session initiates load state using the **JoobConnection BeginLoad** method. Load state is terminated when the **JoobConnection EndLoad** method is called, or when a transaction is terminated (that is, it is committed or rolled back).

When a JADE session is in load state, JADE objects remain locked until load state ends, even if explicitly unlocked using the **JoobContext Unlock** method.

Using load state avoids the overhead of individually unlocking JADE objects, as objects are unlocked together as a single operation. It also avoids repeatedly locking the same objects, as they remain locked until load state ends.

When **EndLoad** is called to end load state, only those locks that were acquired while in load state are unlocked. Locks that were in place before load state was activated remain locked.

Load state affects only locks with transaction duration. Session duration locks are not affected.

You can nest **BeginLoad** calls. A matching **EndLoad** call unlocks only transaction duration locks acquired at the current nesting level. Load state remains in effect until all **BeginLoad** calls have been matched with **EndLoad** calls, or a transaction terminates. Terminating a transaction (for example, by calling **Commit** or **Rollback**) ends load state for all nested levels.

Load state can be of use where the same JADE collection is being accessed repeatedly in a code sequence. However, you should keep in mind that while locks are in effect, access by other JADE sessions can be restricted, particularly if the sessions want to update the objects that get locked.

The following code fragment is a contrived example that illustrates load state.

```
context.Connection.BeginLoad();
C1 c1a = root.AllC1s["a"];
C1 c2b = root.AllC1s["b"];
C1 c2c = root.AllC1s["c"];
context.Connection.EndLoad();
```

Without load state, the collection **root.AllC1s** would be implicitly shared-locked and unlocked three times: once for each indexed access. With the use of load state, the collection is locked and unlocked once only, as it remains locked until load state ends.

Upgrading and Downgrading Locks

This section describes upgrading and downgrading locks.

General

A lock upgrade or downgrade occurs when a JADE session has an object locked, and requests a lock with a different type or duration.

An *upgrade* means that the request is for a type or duration with a greater strength. A *downgrade* means a type or duration of lower strength is requested.

The strength of the various lock types, from lowest to highest, is as follows.

1. Shared
2. Reserve
3. Update
4. Exclusive

For duration, session duration is stronger than transaction duration.

Locks are not released while being upgraded or downgraded, with one exception: when a shared lock is being upgraded to an update lock, as described in the following section.

Changing Lock Type

A type upgrade can queue and potentially time out, causing a **JoobObjectLockedException** to be thrown, if the requested type is not compatible with existing locks. For example, this could happen when upgrading a shared lock to exclusive.

Lock type downgrades will never be queued, as the strength is being lowered so there will be no lock incompatibilities.

When a JADE session is in transaction state, requests to downgrade lock type are ignored. The lock maintains its current type. However, lock types can be upgraded regardless of transaction state.

When a lock type is being upgraded from shared to update, the object is unlocked before the update lock is requested. This happens even if the JADE session is in transaction state, and is the only situation where an object is unlocked while in transaction state. The reason for doing this is to prevent potential deadlocks, as discussed in more detail under "Avoiding Deadlock Exceptions", later in this chapter.

The following code fragment gives examples of upgrading and downgrading lock types.

```
TimeSpan timeOut = TimeSpan.FromSeconds(10);
context.Lock(obj1, LockType.Shared, LockDuration.Transaction, timeOut);
context.Lock(obj1, LockType.Reserve, LockDuration.Transaction, timeOut);
                                // The lock is now upgraded from shared to reserve.
context.Lock(coll, LockType.Exclusive, LockDuration.Transaction, timeOut);

using (System.Data.IDbTransaction tran = context.BeginTransaction())
{
    context.Lock(obj1, LockType.Exclusive, LockDuration.Transaction,
                       timeOut); // The lock type is upgraded to exclusive, as
                                 // locks can be upgraded (but not downgraded)
                                 // when in transaction state.
    foreach (C1 obj2 in coll)
    {
        // The exclusive lock on coll is not downgraded by the implicit shared
        // lock associated with foreach, because transaction state is in effect.
    }
    context.Lock(obj1, LockType.Shared, LockDuration.Transaction, timeOut);
                      // The lock type is not downgraded, but remains as exclusive.
    tran.Commit();    // All transaction duration locks are released.
}
```

Changing Lock Duration

Lock duration can be upgraded only, from transaction to session. The lock duration cannot be downgraded from session to transaction.

A request to downgrade both duration and type is ignored. The lock type for a lock with session duration can be downgraded only by an explicit lock request that specifies session duration.

Lock duration upgrades do not result in queuing, provided they are not accompanied by requests to upgrade the lock type.

JADE keeps track of the lock type associated with a lock's most-recent session duration request. When a JADE session has a session duration lock and issues a request for a higher lock type but of transaction duration, the lock type is upgraded but the duration remains session duration. If automatic unlocking of transaction duration locks then occurs (for example, when committing or rolling back a transaction), the session duration lock is retained but its lock type reverts to the prior value.

Similarly, when a JADE session is in transaction state and a session duration lock with a lower lock type is requested on an existing lock of transaction duration, the lock duration is upgraded to session but the lock type is not downgraded until the transaction is committed or rolled back.

The following code fragment gives examples of changing lock duration.

```
TimeSpan timeOut = TimeSpan.FromSeconds(5);
context.Lock(obj1, LockType.Shared, LockDuration.Transaction, timeOut);
context.Lock(obj1, LockType.Shared, LockDuration.Session, timeOut);
// obj1’s lock is now shared, session duration.
context.Lock(obj2, LockType.Exclusive, LockDuration.Session, timeOut);
context.Lock(obj2, LockType.Exclusive, LockDuration.Transaction, timeOut);
// obj2’s lock remains exclusive, session duration.
context.Lock(obj3, LockType.Shared, LockDuration.Session, timeOut);
using (System.Data.IDbTransaction tran = context.BeginTransaction())
{
    context.Lock(obj3, LockType.Exclusive, LockDuration.Transaction, timeOut);
    // obj3’s lock is upgraded to exclusive, session duration.
    tran.Commit();
    // obj3’s lock reverts to shared, session duration.
    // Session duration locks remain in place.
}
context.Lock(obj1, LockType.Exclusive, LockDuration.Session,
             TimeSpan.FromSeconds(5));
// obj1’s lock type is upgraded to exclusive.
context.Lock(obj2, LockType.Reserve, LockDuration.Session, timeOut);
// obj2’s lock type is downgraded to reserve.
context.Lock(obj2, LockType.Shared, LockDuration.Transaction, timeOut);
// This is ignored, as it is requesting to downgrade both duration and type.
// obj2’s lock remains as reserve, session duration.

context.Unlock(obj3);
context.Unlock(obj2);
context.Unlock(obj1);
```

Collection Locking

JADE collections are composite objects, each comprising a collection header and one or more collection blocks. When JADE collections are being used, it is imperative that all of the composite parts are fully coordinated by being completely up to date and not changing during access. Because of this, whenever collections are accessed and not already locked, they are automatically locked with a shared lock. They get unlocked after the access has completed, unless the JADE session is in transaction state or in load state.

Enumerating JADE Collections

When a JADE collection is traversed using the default enumerator (for example, indirectly in a standard **foreach** statement or directly in conjunction with the **JoobCollection GetEnumerator** method), it remains shared locked until the enumerator is disposed; for example:

```
foreach (C1 obj1 in collA) //Shared lock is now acquired.
{
    //...examine each member...
}
//Shared lock is now released.
```

In other cases, when a JADE collection is traversed (for example, via a LINQ query or in conjunction with **JoobCollection** methods **StartingAtIndex**, **StartingAtKey**, or **StartingAtObject**), the collection is shared locked only for the period when retrieving a snapshot of member references from the associated collection; for example:

```
IEnumerable<C1> query = from C1 obj1 in collA select obj1;
foreach (C1 obj1 in query)
{
    //...collection is not locked within the loop
}

foreach (C1 obj1 in collA.StartingAtIndex(0))
{
    //...collection is not locked within the loop
}
```

The automatic locking of collections is particularly important when transactions are involved. If a JADE collection is accessed when in transaction state, it is shared locked and remains so until the transaction ends. This can hold up other JADE sessions wanting to update the collection, and it can also contribute to deadlocks.

Using update locks can help to prevent JADE sessions from being held up when JADE collections are updated, as update locks are compatible with shared locks. For more details, see "Optimizing Locking", later in this chapter.

Deadlocks

A deadlock is a situation in which two or more JADE sessions are unable to proceed because they end up waiting on locks held by one another.

Simple Deadlocks

The simple deadlock involves two JADE sessions and two JADE objects.

The initial situation is:

- Session A has an exclusive lock on Object 1
- Session B has an exclusive lock on Object 2

The attempted actions are:

- Session A requests a lock on Object 2
- Session B requests a lock on Object 1

The outcome is:

- Session A is now waiting for Session B to unlock Object 2
- At the same time, Session B is waiting for Session A to unlock Object 1
- Neither JADE session can proceed

Indirect Deadlocks

The indirect case involves three or more JADE sessions.

The initial situation is:

- Session A has an exclusive lock on Object 1
- Session B has an exclusive lock on Object 2
- Session C has an exclusive lock on Object 3

The attempted actions are:

- Session A requests a lock on Object 2
- Session B requests a lock on Object 3
- Session C requests a lock on Object 1

The outcome is:

- The sessions cannot proceed because they are all waiting on each other

Single Object Deadlocks

A deadlock can occur with only one object being involved. For example, this can happen in conjunction with lock type upgrades, as follows.

The initial situation is:

- Session A and Session B both have a shared lock on Object 1

The attempted actions are:

- Session A attempts to upgrade the lock type to exclusive
- Session B also attempts to upgrade the lock type to exclusive

The outcome is:

- Neither session can proceed, because they are waiting for each other due to them both having a shared lock

This type of deadlock occurs with the following sort of code sequence.

```
TimeSpan timeOut = TimeSpan.FromSeconds(5);
using ( System.Data.IDbTransaction tran = context.BeginTransaction())
{
    context.Lock(obj1, LockType.Shared, LockDuration.Transaction, timeOut);
    //...
    context.Lock(obj1, LockType.Exclusive, LockDuration.Transaction, timeOut);
    //...
    tran.Commit();
}
```

Two JADE sessions simultaneously executing this sequence using the same JADE object can get in a deadlock situation.

This type of sequence can occur subtly. For example, consider the following pattern where a JADE session checks to see if a JADE collection contains an object, then adds it to the collection if it is not there.

```
using (System.Data.IDbTransaction tran = context.BeginTransaction())
{
    if (!allPeople.Contains(person))
        allPeople.Add(person);
    tran.Commit();
}
```

A deadlock can result if two JADE sessions execute this sequence at the same time with the same collection. Using **Contains** places a shared lock on the collection, and adding the member attempts to upgrade the lock to exclusive (or update). Both JADE sessions can end up with a shared lock on the collection, thereby preventing one another from upgrading.

Deadlock Detection

JADE automatically detects when a direct or indirect deadlock situation would arise due to a lock or lock upgrade request, and throws a **JoobDeadlockException** if so. By default, the exception is thrown for the JADE session requesting the lock.

This JADE session must release locks to allow other JADE sessions waiting to lock objects to continue and then take appropriate action. If the JADE session is in transaction state, it must roll back the transaction in order to unlock objects.

Deadlock Exceptions

By default, the **JoobDeadlockException** is thrown for the JADE session that made the request that caused the deadlock. This can be varied, to throw an exception for *both* JADE sessions involved, or by using a priority to determine which JADE session gets the exception.

Double Deadlock Exceptions

The JADE session that makes the lock request causing a deadlock gets the deadlock exception thrown.

The JADE session that is waiting for the lock held by the session making the request can also have a **JoobDeadlockException** thrown for it, by enabling double deadlock exceptions. This is specified using the following parameter in the JADE initialization file.

```
[JadeServer]
DoubleDeadlockException=true
```

The default value is **false**.

For indirect deadlocks (involving three or more JADE objects), the other sessions involved do not have exceptions thrown for them.

When enabled, double deadlock exceptions apply to *all* deadlocks that occur.

Double deadlock exceptions can be useful when investigating the causes of deadlocks.

Avoiding Deadlock Exceptions

This section describes how to avoid deadlock exceptions.

Lock Order

For deadlocks involving multiple JADE objects, the classic technique for avoiding deadlocks is that each JADE session should lock the objects in the same order. The *direct* and *indirect* examples described earlier in this chapter could not arise if this principle is used. For example, the following two code fragments cause a deadlock, if executed by different JADE sessions at the same time.

```
context.Lock(obj1, LockType.Shared, LockDuration.Transaction,
                      TimeSpan.FromSeconds(5));
```

```
//...
context.Lock(obj2, LockType.Exclusive, LockDuration.Transaction,
                    TimeSpan.FromSeconds(5));
//...
context.Unlock(obj2);
context.Unlock(obj1);


context.Lock(obj2, LockType.Shared, LockDuration.Transaction,
             TimeSpan.FromSeconds(5));
//...
context.Lock(obj1, LockType.Exclusive, LockDuration.Transaction,
             TimeSpan.FromSeconds(5));
//...
context.Unlock(obj1);
context.Unlock(obj2);
```

The first JADE session cannot get the exclusive lock on **obj2** because the second session has a shared lock on it. The second JADE session cannot get the exclusive lock on **obj1** because the first session has a shared lock on it.

The deadlock cannot occur if the second code sequence locks the objects in the same order as the first, as follows.

```
context.Lock(obj1, LockType.Exclusive, LockDuration.Transaction,
             TimeSpan.FromSeconds(5));
//...
context.Lock(obj2, LockType.Shared, LockDuration.Transaction,
             TimeSpan.FromSeconds(5));
//...
context.Unlock(obj2);
context.Unlock(obj1);
```

The first JADE session to lock **obj1** blocks the other, preventing it from locking **obj2** and setting up the deadlock.

Lock ordering issues are not always obvious, as locks are commonly implicitly acquired. For example, accessing a JADE collection implicitly acquires a shared lock, and updating an object implicitly acquires an update or exclusive lock.

Reserve Locks

Reserve locks can be a useful way to avoid single object deadlocks due to lock upgrade requests. For example, the following code fragment from an earlier example in this chapter can cause a JoobDeadlockException to be thrown if executed by two JADE sessions at the same time with the same object, due to both wanting to upgrade a lock on the **allPeople** collection from shared to exclusive:

```
using (System.Data.IDbTransaction tran = context.BeginTransaction())
{
    if (!allPeople.Contains(person))
        allPeople.Add(person);
    tran.Commit();
}
```

However, if the code initially locks the collection with a reserve lock instead of the implicit shared lock, the deadlock situation will not arise; for example:

```
using (System.Data.IDbTransaction tran = context.BeginTransaction())
{
    context.Lock(allPeople, LockType.Reserve, LockDuration.Transaction,
                 TimeSpan.FromSeconds(5));
    if (!allPeople.Contains(person))
        allPeople.Add(person);
    tran.Commit();
}
```

The first JADE session to execute this code sequence acquires the reserve lock. Other JADE sessions executing this sequence queue until the object is unlocked; that is, when the transaction is committed, as a reserve lock is not compatible with other reserve locks. The lock, therefore, can be readily upgraded to exclusive, and the single object deadlock will not occur.

Using an initial reserve lock as opposed to an exclusive lock has the benefit of not blocking other JADE sessions from obtaining shared locks. This could be significant in cases where there is a reasonable time gap between when the object is first locked and when it is updated. However, when that gap is very short, as in the above code example, an exclusive lock may be preferable, as it avoids the overhead of upgrading the lock.

Update Locks

Deadlock exceptions due to lock upgrades can be avoided by using update locks instead of exclusive locks, as an object gets unlocked before its lock is upgraded from shared to update (as mentioned in "Upgrading and Downgrading Locks", earlier in this chapter). For example, you could change the above code sequence to the following.

```
using (System.Data.IDbTransaction tran = context.BeginTransaction())
{
    bool retry = false;
    do
    {
        if (!allPeople.Contains(person))
        {
            if (retry)
                retry = false;
            else
            {
                try
                {
                    context.Lock(allPeople, LockType.Update,
                                            LockDuration.Transaction,
                                            TimeSpan.FromSeconds(2));
                }
                catch (JoobInterveningUpdateException)
                {
                    retry = true;
                    continue;
                }
            }
            allPeople.Add(person);
        }
    }
    while (retry);

    tran.Commit();
}
```

A deadlock does not occur when two JADE sessions execute this code sequence at the same time. Even though both JADE sessions can end up with shared locks on the collection, because the collection gets temporarily unlocked, one of the sessions acquires the update lock, allowing it to proceed to transaction completion and unlock the collection, then followed by the other session.

Note that this code fragment has to deal with a **JoobInterveningUpdateException**, which can be thrown if another JADE session updates the collection before the update lock is acquired (while it is unlocked). To do this, the collection is explicitly locked and a catch block is used to indicate the operation should be retried. The **JoobInterveningUpdateException** is thrown after the update lock is acquired, so the object does not need to be locked again. An explicit lock is used, because if an exception is thrown while an implicit lock is being attempted for inverse maintenance, the transaction is rolled back before the catch block is executed. This avoids having to repeat the whole transaction.

For more details, see "Exceptions", later in this chapter.

Investigating Deadlocks

When a deadlock exception is thrown, the **JoobDeadlockException** object contains properties that allow analysis of the deadlock situation. This information includes a reference to the object for which the lock was requested and a reference to the object for which the other JADE session involved with the deadlock was waiting. (For details, see "JoobDeadlockException", later in this chapter.)

For deadlocks involving more than two JADE sessions, there is no direct way to determine the JADE session that was requesting the lock on the object that the current JADE session has locked. However, to aid analysis, you can configure the JADE initialization file to cause deadlock exceptions to be thrown for both JADE sessions involved. (For details, see "Double Deadlock Exceptions", earlier in this chapter.)

You can obtain additional information about deadlocks by configuring the [Log] section in the JADE initialization file to enable queue and lock tracing in the JADE message log files.

Exceptions

The three main exceptions related to locking are as follows.

- **JoobObjectLockedException** – object lock request timed out.
- **JoobDeadlockException** – deadlock detected.
- **JoobInterveningUpdateException** – the object was updated before the lock upgrade completed.

JoobObjectLockedException

A **JoobObjectLockedException** is thrown when a lock request cannot be granted within the specified lock timeout interval, due to incompatible locks held by other JADE sessions.

The **JoobObjectLockedException** object properties that indicate details about the lock request and the JADE session that had the object locked at the time are listed in the following table.

Property	Type	Description
LockTarget	ObjectId	The object requested to be locked
LockType	LockType	The requested lock type
LockDuration	LockDuration	The requested lock duration - **Transaction** or **Session**
LockTimeout	TimeSpan	The requested lock timeout period
TargetLockedBy	ObjectId	The JADE session that had the object locked

If required, you can repeat an explicit lock request, by using a **catch** block and a loop. This provides a way to extend the lock timeout period in certain circumstances; for example:

```
bool locked = false;
int  retries = 0;
while ( !locked )
{
    try
    {
        context.Lock(obj1, LockType.Update, LockDuration.Transaction,
                           TimeSpan.FromSeconds(4) );
        locked = true;
    }
    catch (JoobObjectLockedException ex)
    {
        if ( ++retries > 5 )
             throw ex;
        Console.WriteLine("Object{0} locked by {1} - retrying", obj1.ToString(),
                          ex.TargetLockedBy.ToString());
    }
}
```

The exception can be avoided, by using the **JoobContext TryLock** method. This method returns **true** or **false**, to indicate the outcome of the lock request, instead of throwing an exception if the request times out; for example:

```
if ( !context.TryLock(obj1, LockType.Exclusive, LockDuration.Transaction,
                      TimeSpan.FromSeconds(4)) )
{
    Console.WriteLine("Could not lock {0}", obj1.ToString());
}
```

Note Although the **TryLock** method does not result in a **JoobObjectLockedException** being thrown, a **JoobDeadlockException** can still occur.

A JoobObjectLockedException can be thrown as a result of an implicit lock request made by JADE. Implicit lock requests happen when JADE collections are accessed, or objects get implicitly locked for updating, including during inverse maintenance to automatically update collections. For implicit lock requests, the default lock timeout period is specified in the **ServerTimeout** parameter in the [JadeServer] section of the JADE initialization file.

For straightforward implicit lock requests, it may be appropriate to catch the exception and retry the operation that resulted in the implicit lock request, if a longer timeout period was wanted. However, when a **JoobObjectLockedException** is thrown during inverse maintenance (for example, when changing the collection parent reference), the transaction is automatically rolled back, so the operation cannot be retried without reprocessing the whole transaction. The transaction is rolled back in order to maintain referential integrity, which can be compromised if inverse maintenance is not completed.

A **catch** block can check if the transaction has been rolled back by checking the **JoobConnection IsTransactionActive** property; for example:

```
try
{
    Member member = new Member();
    member.Id = 99;
    member.MyOwner = owner;
    // ...
}
catch
```

```
    {
        if ( !context.Connection.IsTransactionActive )
        {
            Console.WriteLine("Transaction was automatically rolled back");
        }
        else
            tran.Rollback();
    }
```

As an alternative, the code can avoid lock exceptions on implicit locks by instead explicitly locking collections before the operation involving inverse maintenance.

When update locks are being used, a lock exception can be thrown when committing a JADE transaction. This happens if an update lock cannot be upgraded to an exclusive lock. In this circumstance, the transaction will have been rolled back. A **catch** block can be used to detect this, for example:

```
    using (System.Data.IDbTransaction tran = context.BeginTransaction())
    {
        // ...
        try
        {
            tran.Commit();
        }
        catch
        {
            Console.WriteLine("Could not commit transaction due to an exception");
        }
    }
```

JoobDeadlockException

A **JoobDeadlockException** is thrown when JADE detects that a lock request would result in a deadlock condition. The lock request can be explicit or implicit, including when committing a transaction involving update locks.

At a minimum, the JADE session must abandon the lock request and release the lock that was blocking the other session. If the JADE session is in transaction state, the transaction must be rolled back in order to release the lock. Rolling back the transaction releases all transaction duration locks.

The **JoobDeadlockException** object properties that indicate details about the deadlock are listed in the following table.

Property	Type	Description
LockTarget	ObjectId	Identifies the object requested to be locked
LockType	LockType	The requested lock type
LockDuration	LockDuration	The requested lock duration
LockTimeout	TimeSpan	The requested lock timeout period
TargetLockedBy	ObjectId	Identifies the JADE session that has the object locked
ObtainedLock	ObjectId	Identifies the object that the requesting JADE session has locked, which caused the deadlock

The following code fragment is an example of displaying **JoobDeadlockException** information.

```
catch ( JoobDeadlockException dlException )
{
    Console.WriteLine("Requested object {0}", dlException.LockTarget.ToString());
    Console.WriteLine("Requested type {0}", dlException.LockType.ToString());
    Console.WriteLine("Locked by {0}", dlException.TargetLockedBy.ToString());
    Console.WriteLine("Obtained lock {0}", dlException.ObtainedLock.ToString());
    //...
}
```

JoobInterveningUpdateException

A **JoobInterveningUpdateException** is thrown when a JADE session has upgraded a lock on an object from shared to update, but another JADE session has updated the object in the interim when it was unlocked.

The exception is thrown after the update lock has been acquired. An application therefore has the option to ignore the exception and continue execution, if appropriate; for example:

```
TimeSpan timeout = TimeSpan.FromSeconds(5);
using (System.Data.IDbTransaction tran = context.BeginTransaction())
{
    context.Lock(collA, LockType.Shared, LockDuration.Transaction, timeout);
    // ...
    try
    {
        context.Lock(collA, LockType.Update, LockDuration.Transaction, timeout);
    }
    catch (JoobInterveningUpdateException ex)
    {
        // Ignore the exception: the code does not depend on previously
        // checked values (that may have changed).
    }
    //...
}
```

An alternative would be to resume execution from an earlier point in order to refresh any properties of the object (that is now update locked) that may have changed, without needing to roll back the current transaction.

The following code fragment gives an example of how this could be done.

```
bool    retryNeeded;
context.SetImplicitUpdatingLockType(LockType.Update);
using (System.Data.IDbTransaction tran = context.BeginTransaction())
{
    do
    {
        retryNeeded = false;
        try
        {
            //Calling TryGetValue acquires a shared lock on the collection.
            Person existingPerson;
            if ( !allPeople.TryGetValue( person.Id, out existingPerson) )
            {
                //Adding upgrades the lock from shared to update, which
                //unlocks the collection in the interim.
                allPeople.Add(person);
```

```
                    }
                }
                catch ( JoobInterveningUpdateException ex )
                {
                     retryNeeded = true; //Need to check if the person has just been
                                         //added to the collection by another process.
                }
            }
            while (retryNeeded);
        }
```

In this example, if a **JoobInterveningUpdateException** is thrown, the JADE session needs to check again to see if the person is in the collection, as it could have been added by another JADE session while the collection was unlocked. Note that when retrying, the collection is locked with an update lock.

Object Volatility

Object volatility refers to the frequency with which a JADE object is updated. This, in turn, can be used to customize locking behavior and provide optimizations.

JADE defines the following three types of volatility.

- Volatile, for objects that are regularly updated.
- Stable, for objects that are infrequently updated.
- Frozen, for objects that are almost never updated.

JADE objects are volatile, by default. Volatility can be specified at the JADE class level. There are also methods that enable you to alter and interrogate the volatility for individual JADE objects.

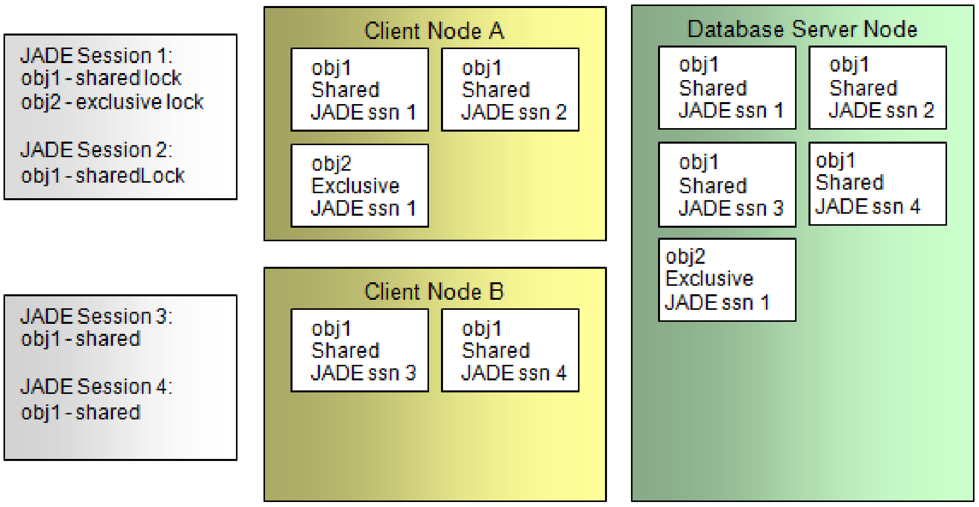
The Schema Inspector displays class and object volatility by default, which enables you to check whether objects and collections are set to stable or frozen without having to write code to determine the volatility state of an object or collection.

Volatile Objects

A volatile object is one that is updated reasonably frequently. Normal locking is used on volatile objects.

A volatile object is recorded as locked by any JADE sessions that have it locked until they unlock it, and whenever locked, the latest edition of the object is fetched from the database server.

The following diagram shows the locks recorded for volatile objects.



Every lock is recorded locally on the client node and remotely on the database server node.

Stable Objects

A stable object is one that is mostly read, and only infrequently updated.

For stable objects, each node retains a generic *node shared lock*, recorded on the server node and the local client node, even after JADE sessions have unlocked the object. Shared lock entries for individual JADE sessions are recorded only on the local client node. The node's shared lock is released when a JADE session requests an exclusive lock on the object, when the object is swapped out of the client node's object cache, or when the client node terminates.

Using stable objects avoids the overhead of communication with the database server node to acquire a shared lock and retrieve the object from the JADE database. This is because while a node lock is in place, the lock can be handled locally, and the copy of the object in cache is the latest edition, because the object cannot be updated while the client node's shared lock is in place.

When an exclusive lock is requested on a stable object, all client nodes that have node shared locks for that object are asked to release the lock. The client nodes release the lock when no JADE sessions on that node have the object locked.

To prevent the exclusive lock request from being held up by other shared lock requests, further lock requests are blocked until after the exclusive lock has been acquired and released (or the request times out).

Requests for reserve locks and update locks are handled normally. Node locks are not involved, as these lock types are compatible with shared locks. In addition, requests for shared locks of session duration are handled normally, and do not involve node locks.

An example of a good use of stable objects would be for JADE collections that are frequently enumerated but infrequently updated.

The trade-off is that although stable objects optimize the acquisition of shared locks, acquiring exclusive locks involves extra overhead, as all client nodes that are holding node locks need to be requested to release the locks first.

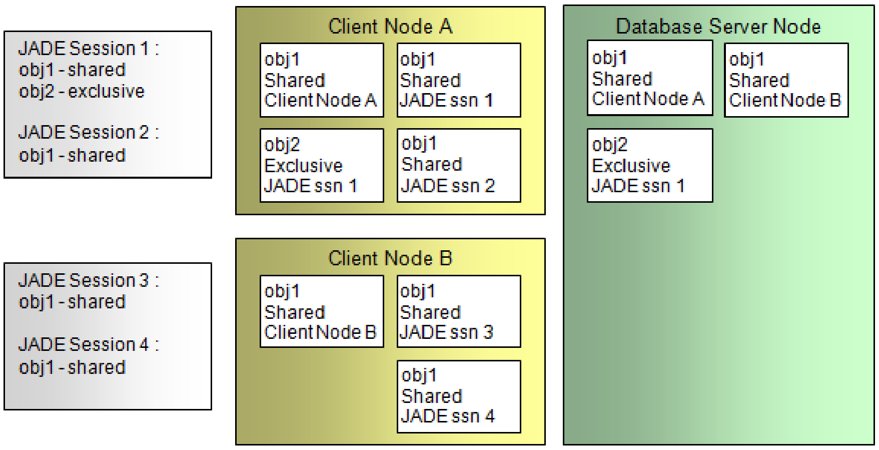
Another factor to consider is that using stable objects can result in an increase in the number of locks held in the server node's lock tables, because shared locks tend to be retained for longer. For very large numbers of concurrent locks (10,000 or more), it may be beneficial to increase the value of the **PersistentLockHashSize** parameter the **PersistentLockHashSize** parameter in the [JadeClient] and [JadeServer] section of the JADE initialization file. For details, see the *JADE Initialization File Reference*.

Node locks are identifiable when lock information is displayed by the JADE Monitor, as node locks have the **kind** property set to **LockKind.Node**.

Local shared locks on stable objects (that is, locks held on client nodes by JADE sessions where a node lock is involved) have the **kind** property set to **LockKind.Local**.

Other locks have the **kind** property set to **LockKind.Normal**.

The following diagram shows the locks recorded for stable objects.



This example illustrates that shared locks for individual JADE sessions are recorded locally on the session's client node, whereas a node lock is recorded on the client node and the database server node. Exclusive locks are recorded on the client node and database server node, in the same manner as volatile objects.

Note For convenience, node locks use the client node's JADE background session as the session that has the lock.

Frozen Objects

A frozen object is one that is updated rarely. In fact, when a JADE object is frozen, it cannot be updated until the volatility is changed, and you should do this only when the system is unavailable to general users. Examples of good candidates for being frozen include code tables used for interpreting abbreviations, or JADE objects that do not change after they are created (for example, commercial events or archive data).

For frozen objects, shared and reserve lock requests are ignored. There is no need for the lock to be applied, as frozen objects cannot be updated and therefore a consistent view of the data is guaranteed. In addition, a frozen object in cache is always the latest edition, avoiding the need to fetch the object from the database server node.

Update locks are not allowed on frozen objects.

Exclusive locks on frozen objects are actioned.

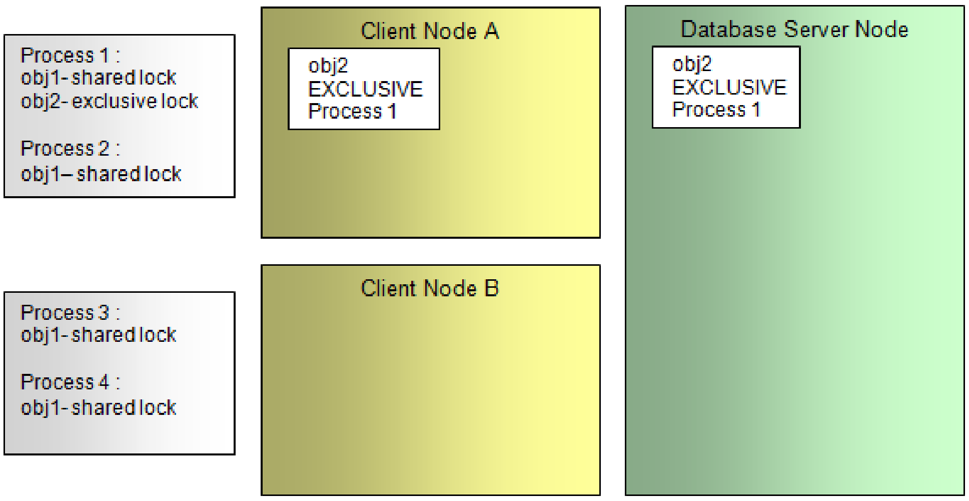
Special action is required in order to update an object that is currently frozen. The object must first be thawed, by changing the volatility to **volatile** or **stable**, using the **JoobObject SetVolatility** method described under "Specifying and Changing Object Volatility", later in this chapter.

Thaw a frozen object with caution, as other JADE sessions using the object may temporarily not know that the object has been updated, and could even be assuming that it cannot change as they have requested shared locks on the object (which will have been ignored if the object was frozen).

We recommend two techniques for changing the volatility of frozen objects. The first is to thaw and update frozen objects when you can be sure that no other JADE sessions can be using the object and that there are no copies in local caches; for example, when the system is run in single user mode or it is offline and unavailable to general users.

The second technique is to change the volatility conditionally; that is, to use methods that change the volatility of a frozen object only if the object's class is not currently in use by another JADE session. For details, see the following section.

The following diagram shows an example of locks held when requesting locks on frozen objects **obj1** and **obj2**.



This example illustrates that exclusive lock requests are honored for frozen objects, whereas shared lock requests are ignored.

Specifying and Changing Object Volatility

You can specify volatility at the individual object level or at a JADE class level.

Class Volatility

Class volatility is specified in the class definition in the JADE Class Browser. The volatility options are **volatile**, **stable**, or **frozen**. When you specify volatility at the class level, you specify the volatility of all future instances of a class. By default, class instances are volatile.

The volatility specified for a class is not inherited by subclasses. In addition, changing a class's volatility does not affect existing instances. To change the volatility of existing instances, you must use the methods outlined in the following section.

Setting the class volatility to frozen is a good option for objects that do not change once they are created; for example, records of sales and purchases.

Note The initial volatility for exclusive collections is as defined for the collection's class. The volatility of the object containing the collection is not considered.

Individual Object Volatility

For individual JADE objects, the **JoobObject Volatility** property indicates the object's volatility. It can have one of the following values.

- JoobObjectVolatility.Volatile
- JoobObjectVolatility.Stable
- JoobObjectVolatility.Frozen

An object's volatility can be changed using the **JoobObject SetVolatility** method, which has the following signature.

```
void SetVolatility(JoobObjectVolatility volatility, bool conditional);
```

The **conditional** parameter specifies whether changing the volatility of a frozen object should be conditional or unconditional. If conditional, a **JoobCannotThawClassInUseException** is thrown if the object's class is in use. The **conditional** parameter is ignored when changing the volatility of volatile or stable objects.

Unconditionally change the volatility of a frozen object only when it can be assured that no other JADE sessions are using the object.

Note If production mode is set, volatility can be conditionally changed only in single user mode. If attempted in multiuser mode, a **JoobFeatureNotAvailableException** is thrown.

The following example demonstrates defining a collection to be stable.

```
Company company = null;
//...
company.AllStaff.SetVolatility(JoobObjectVolatility.Stable, false);
```

The following example demonstrates checking if an object is stable.

```
if (company.AllStaff.Volatility == JoobObjectVolatility.Stable)
    Console.WriteLine("{0} is stable", company.AllStaff.ToString());
```

The following example demonstrates thawing a frozen object conditionally.

```
company.Codes.SetVolatility(JoobObjectVolatility.Volatile, true);
```

The Schema Inspector displays class and object volatility by default, which enables you to check whether objects and collections are set to stable or frozen without having to write code to determine the volatility state of an object or collection.

Frozen Files and Partitions

A convenient way to freeze a large number of JADE objects is to freeze a database file or partition. In effect, this makes all objects read from that file or partition *frozen*. This reduces the number of lock and unlock requests, and brings about the same overhead savings as if the objects had been frozen individually.

Frozen partitions can be used for a class where the partition for instances is determined by a date attribute. Partitions containing old, historical data could be marked frozen. Specific examples include sales or orders, banking transactions, test results, meter readings, and so on.

Use the JADE Database Administration utility to freeze files. The following example shows how to freeze the first (oldest) partition of **MyFile**.

```
jdbadmin ini=iniFile path=system action=Freeze file=MyFile partition=1
```

Freezing a file or partition overrides any specified object volatility for all instances defined in that file or partition. This means a volatile or stable object stored in a frozen file or partition is treated as a frozen object when the object is read. The objects can be reverted to their actual volatility, by thawing the file or partition.

Use the JADE Database Administration utility to thaw files. The following example shows how to thaw the first (oldest) partition of **MyFile**.

```
jdbadmin ini=iniFile path=system action=Thaw file=MyFile partition=1
```

For more details, see "Using the JADE Database Administration Utility", in Chapter 2 of the *JADE Database Administration Guide*.

Note Modify the backup strategy to take advantage of the fact that frozen files or partitions cannot be updated, so do not need to be backed up as frequently.

Freezing User Schema Files

The **_userscm**, **_userxrf**, and **_userint** user schema database files contain the metaschema entities that describe all user classes, properties, applications, and so on. These files are not updated while a JADE session is running. In fact, they change only when a schema is deployed. Obtain the performance benefits associated with frozen objects by freezing these files.

Use the **FreezeSchemaFiles** command of the JADE Database Administration utility to freeze the user schema database files, as follows.

```
jdbadmin ini=iniFile path=system action=FreezeSchemaFiles
```

You must thaw user schema database files before deploying a schema.

Use the **ThawSchemaFiles** command of the JADE Database Administration utility to thaw the user schema database files, as follows.

```
jdbadmin ini=iniFile path=system action=ThawSchemaFiles
```

Chapter 3 Introductory Tutorial to JADE .NET

This chapter covers the following topics.

Overview

In this tutorial, the banking system that is built by attendees on the five-day JADE Developer's course is exposed to .NET. You can download the schema files for the completed banking system (**JADE-Banking-Schema**) from https://github.com/jadesoftwarenz.

The following sections explain in detail how the exposure is created using the JADE Exposure wizard and how a WPF application is built in Visual Studio to use the exposed classes to add customers into the JADE database.

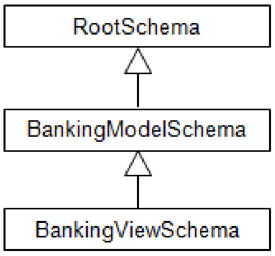
JADE Banking System

This section describes the JADE system that is built during the five-day JADE Developer's course. It is based on customers of a bank and their bank accounts.

One of the aims of the tutorial is to demonstrate the functionality of JADE .NET with an exposure that is as small and simple as possible. Consequently, only part of this system is exposed to .NET. To limit the size of the exposure, the bank account classes are not exposed.

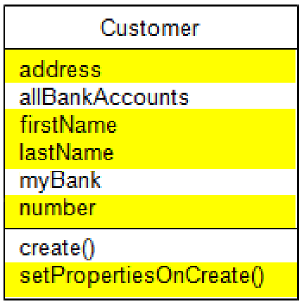
Model-View Separation

In the design of the banking system, the *model* classes (mainly the classes for objects in the persistent database) are in the **BankingModelSchema**. They are separated from the *view* classes (mainly the forms and application definitions), which are in the **BankingViewSchema**.



Customer Class

The **address**, **firstNames**, **lastName**, and **number** properties are part of the exposure because they will be accessed from .NET.



The **allBankAccounts** reference is not exposed, in accordance with the decision to exclude bank account classes from the exposure.

The **myBank** reference is not exposed because it refers to the root object, which is the same for all **Customer** objects and therefore not semantically important. The **myBank** reference can be regarded as part of the JADE implementation in that when it is set, a **Customer** object is added to the root object's **allCustomers** collection through the automatic inverse reference maintenance that is triggered. The setting of the **myBank** reference is encapsulated in the **setPropertiesOnCreate** method.

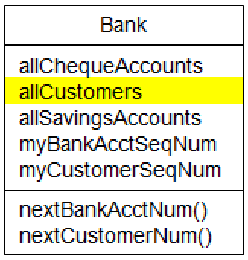
The **setPropertiesOnCreate** method is exposed. If this method was not exposed, you would need to expose the **myBank** reference, as explained previously.

Note When you expose a JADE method to .NET, you do not need to expose the properties and methods that are accessed in that method. This is in line with the practice of exposing an interface but hiding the implementation.

The **create** method is not exposed. However, when a **Customer** object is created in .NET code, the JADE constructor is executed causing a unique value to be assigned to the **number** property.

Bank Class

The **Bank** class is the root object class. An important function of a root object is to provide access to collections of other objects from the model classes. In this tutorial, only the **allCustomers** collection is exposed to .NET.



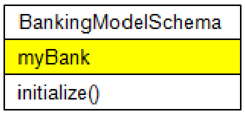
Note When you run the exposure wizard, selecting a reference automatically selects the type of that reference; for example, selecting **allCustomers** automatically exposes the **CustomersByLastName** dictionary class; that is, you do not have to decide which collection classes to select in the exposure wizard.

The properties and methods related to sequence numbers are not exposed, even though they are used by the **create** method of the **Customer** class to generate a unique customer number.

Application Subclass

One of the uses of an application subclass in a JADE system is to hold a reference to the root object, which is an important object in that it enables access to collections of other model objects. The root object reference, which is a key part of the exposure, is **myBank**.

BankingModelSchema is the **Application** subclass in the following diagram.



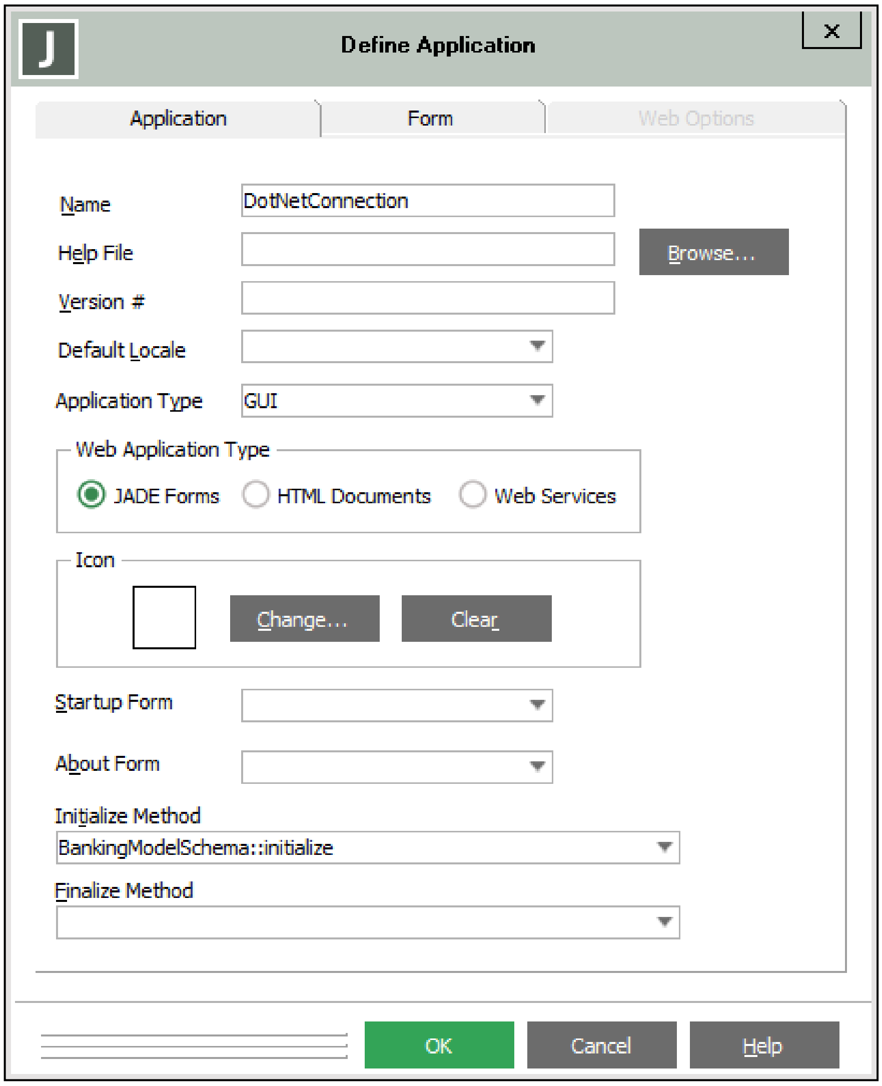
It is not necessary to include the **initialize** method as part of the exposure. However, the **initialize** method should be specified as the startup method in the definition of the client application that connects the .NET application to the JADE database server.

Defining the JADE Connection Application

A JADE connection application is a client application that is started by the .NET application when the .NET application creates a **JoobContext** object for the first time.

The JADE connection application connects the .NET application to the JADE database server. The application and schema name of the JADE configuration application to be started are specified in the **app.config** file of the .NET application. An **app.config** file is generated by the exposure wizard. For more details about application configuration files, see "Application Configuration File", in Chapter 6.

You can define an application for the connection between .NET and JADE, as shown in the following image.


Define Application

Application | Form | Web Options

Name: DotNetConnection
Help File:
Version #:
Default Locale:
Application Type: GUI
Web Application Type: [✓ JADE Forms] HTML Documents Web Services
Icon
Startup Form:
About Form:
Initialize Method: BankingModelSchema::initialize
Finalize Method:

This application is defined in the **BankingViewSchema**.

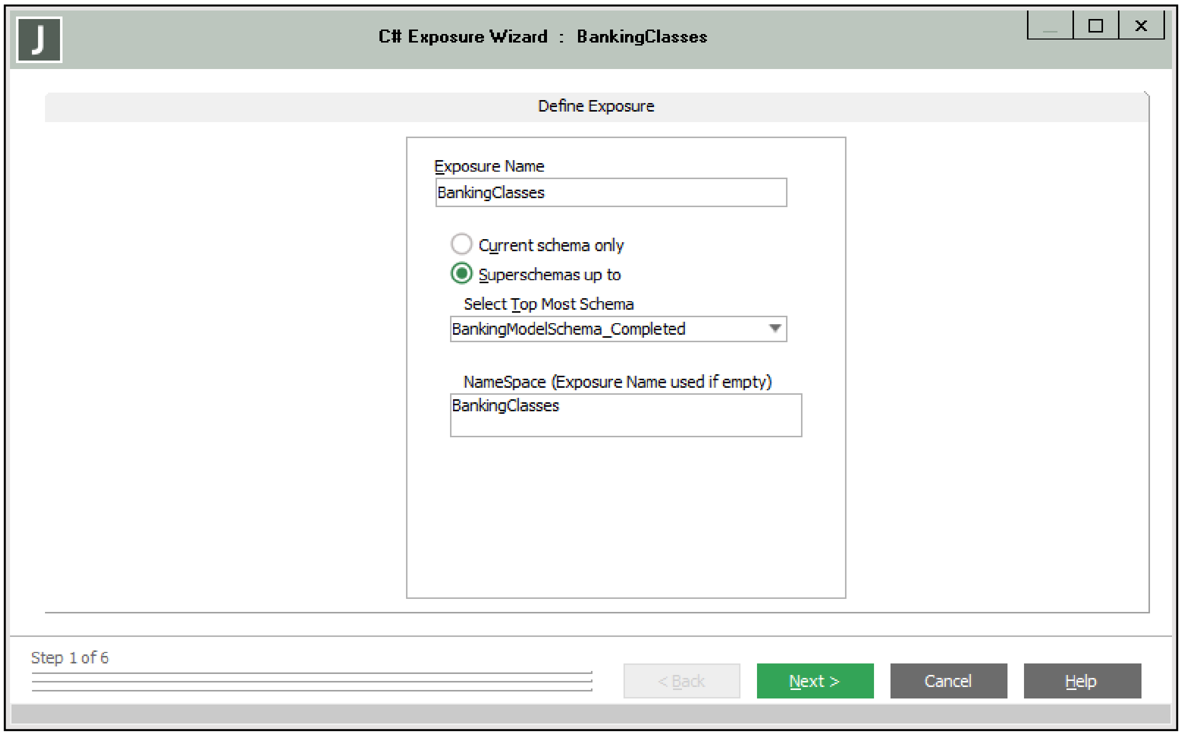
Note The **initialize** method sets a reference on the **app** object to the root object of the banking system in the same way as for any other JADE application. For details about how a .NET application accesses the **app** object, see "Accessing JADE System Objects", in Chapter 7.

Defining the C# Exposure

The following instructions describe how you can create a C# exposure for classes in the banking system. For more details, see "Chapter 17 - Using the C# Exposure Wizard", in the *JADE Development Environment User's Guide*.

1. From the Browse menu in the **BankingViewSchema**, select Exposures and then select the C# tab. Right-click and add an exposure.

2. On the Define Exposure step of the wizard, name the exposure **BankingClasses** and then select schemas up to the **BankingModelSchema**.



C# Exposure Wizard : BankingClasses

Define Exposure

Exposure Name: BankingClasses

Current schema only [✓ Superschemas up to]

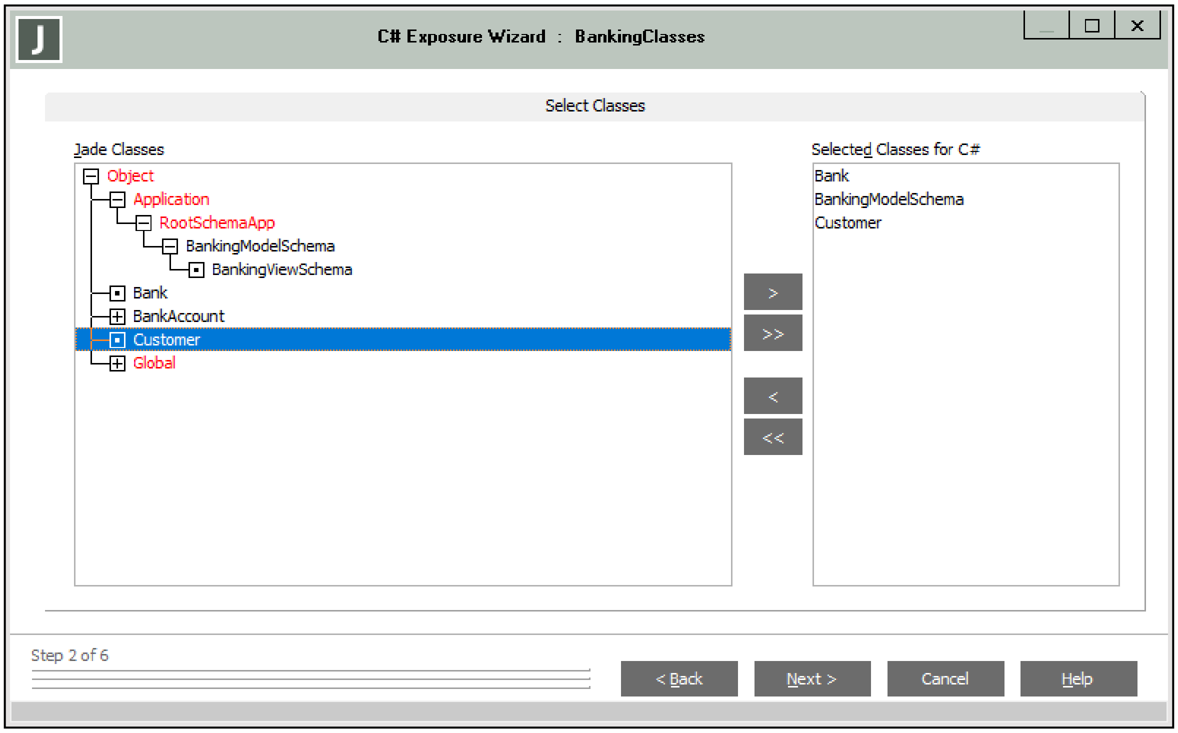
Select Top Most Schema: BankingModelSchema_Completed

NameSpace (Exposure Name used if empty): BankingClasses

Step 1 of 6

3. On the next step of the wizard, select the following classes.

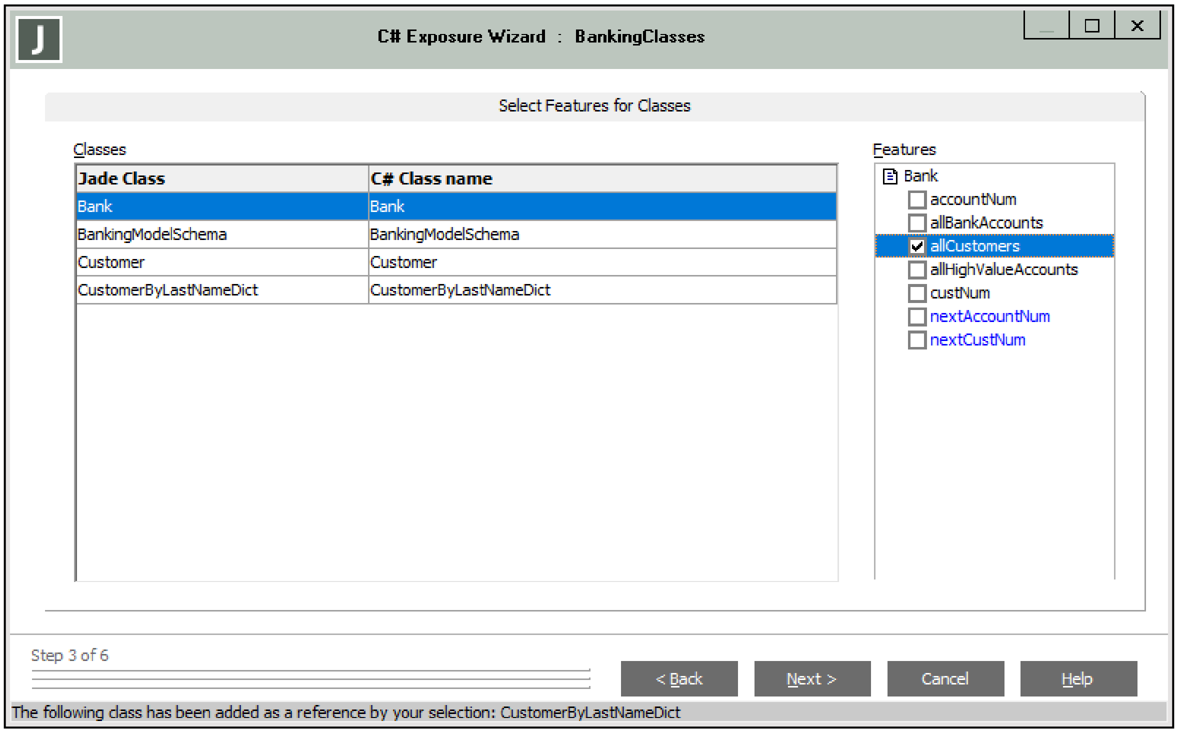
   - **Bank**, to enable the application to access the root object's **allCustomers** collection
   - **BankingModelSchema**, to enable the root object to be accessed using the **myBank** reference
   - **Customer**, because the application will create customers



Selected Classes for C#
Bank
BankingModelSchema
Customer

You do not need direct access to this class from your C# application code. For simplicity in this tutorial, the **BankAccount** classes are not included.

4. On the Select Features step of the wizard, for the **Bank** class, select the **allCustomers** collection.



C# Exposure Wizard : BankingClasses

Select Features for Classes

Classes

Jade Class	C# Class name
Bank	Bank
BankingModelSchema	BankingModelSchema
Customer	Customer
CustomerByLastNameDict	CustomerByLastNameDict

Features

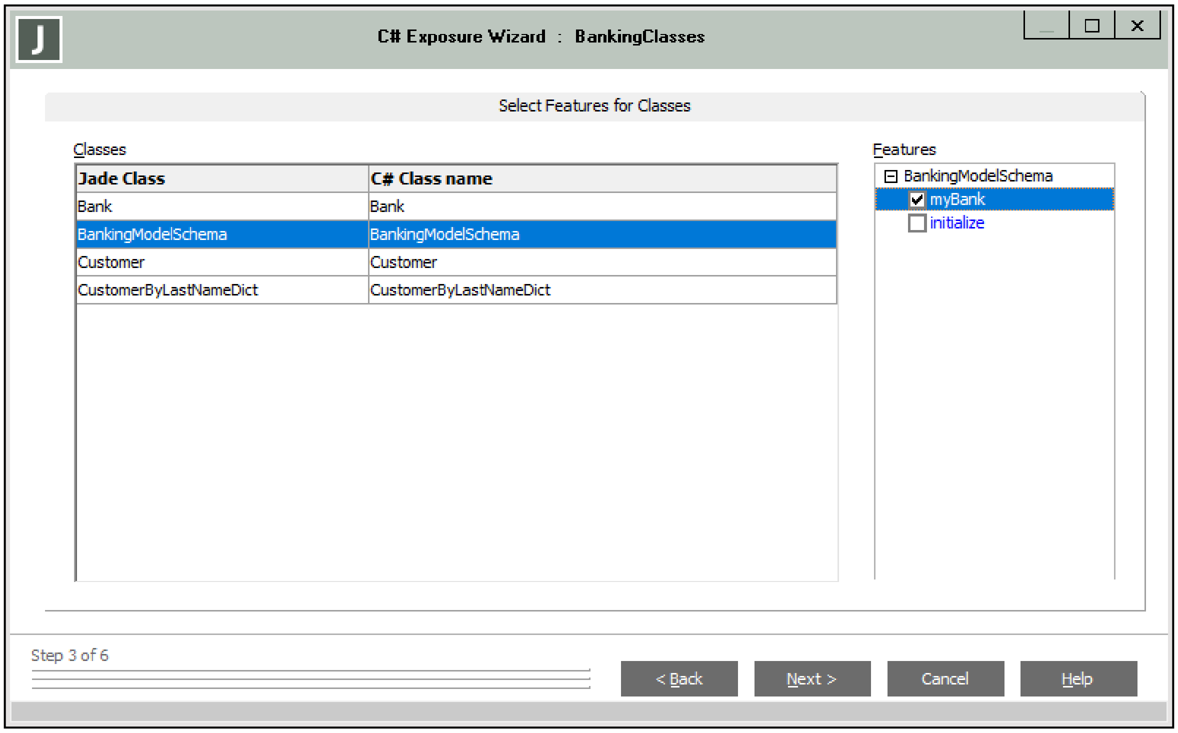
Bank
- [ ] accountNum
- [ ] allBankAccounts
- [x] allCustomers
- [ ] allHighValueAccounts
- [ ] custNum
- [ ] nextAccountNum
- [ ] nextCustNum

Step 3 of 6

The following class has been added as a reference by your selection: CustomerByLastNameDict

Note When you select the **allCustomers** collection, the **CustomerByLastName** collection class becomes part of the exposure. You never need to select the class of reference explicitly when using the C# Exposure wizard.

5. For the **BankingModelSchema**, which is an **Application** subclass, select the **myBank** reference to the root object.


C# Exposure Wizard : BankingClasses

Select Features for Classes

Classes

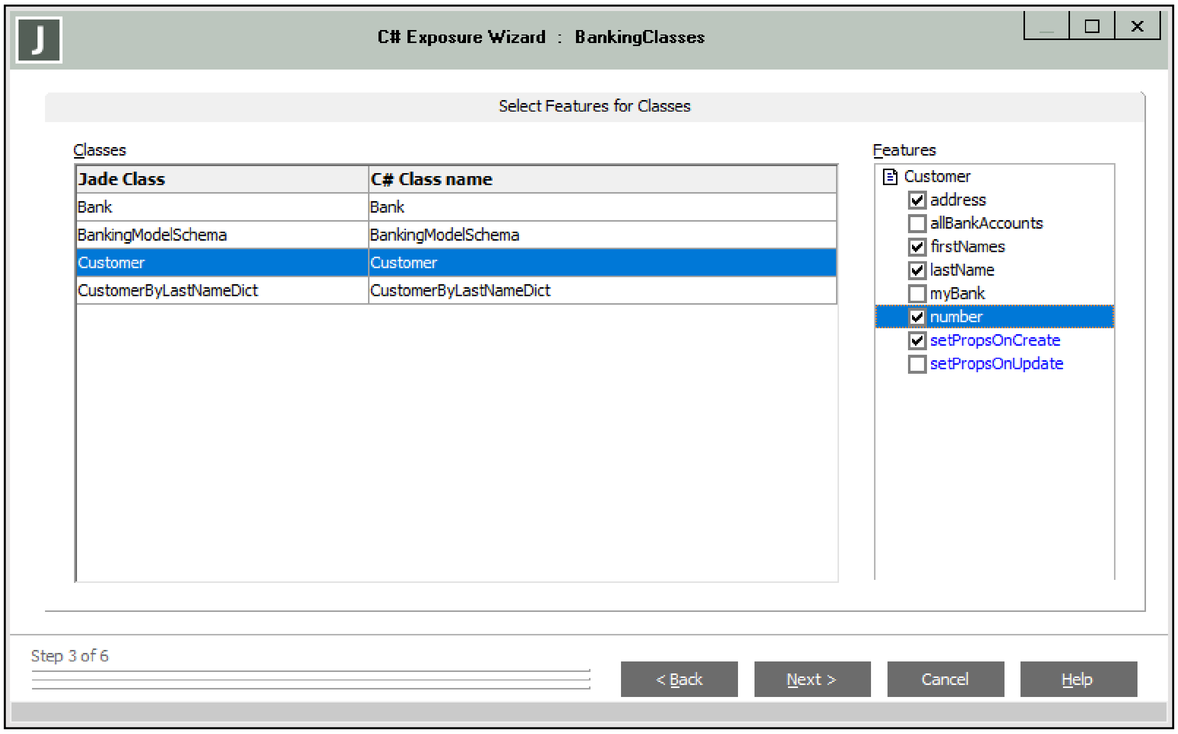
Jade Class	C# Class name
Bank	Bank
BankingModelSchema	BankingModelSchema
Customer	Customer
CustomerByLastNameDict	CustomerByLastNameDict

Features

BankingModelSchema
- [✓] myBank
- [ ] initialize

Step 3 of 6

6. For the **Customer** class, select the **setPropertiesOnCreate** method along with the **address**, **firstNames**, **lastName**, and **number** properties so that it can be invoked from your C# code.


C# Exposure Wizard : BankingClasses

Select Features for Classes

Classes

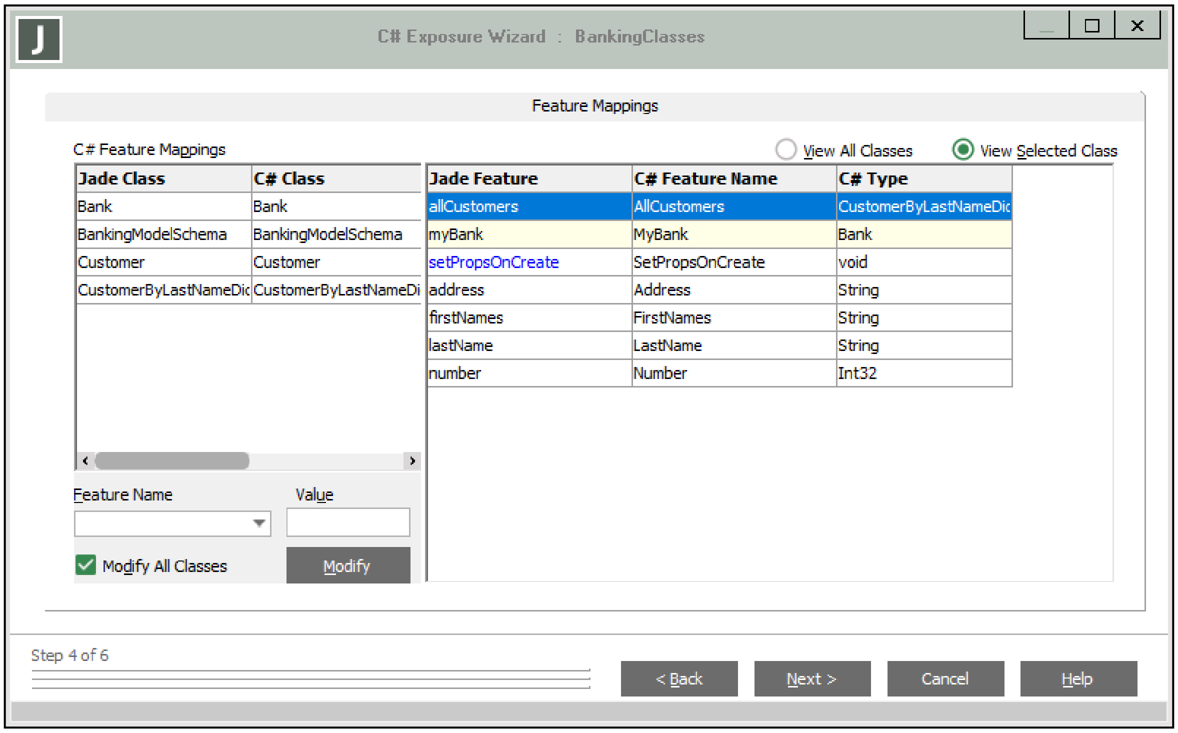
Jade Class	C# Class name
Bank	Bank
BankingModelSchema	BankingModelSchema
Customer	Customer
CustomerByLastNameDict	CustomerByLastNameDict

Features

Customer
- [✓] address
- [ ] allBankAccounts
- [✓] firstNames
- [✓] lastName
- [ ] myBank
- [✓] number
- [✓] setPropsOnCreate
- [ ] setPropsOnUpdate

Step 3 of 6

7. The Feature Mappings step of the wizard enables you to specify different names for the exposed classes, properties, and methods. Accept the default names without change.



View All Classes [✓ View Selected Class]

Jade Class	C# Class
Bank	Bank
BankingModelSchema	BankingModelSchema
Customer	Customer
CustomerByLastNameDic	CustomerByLastNameDi

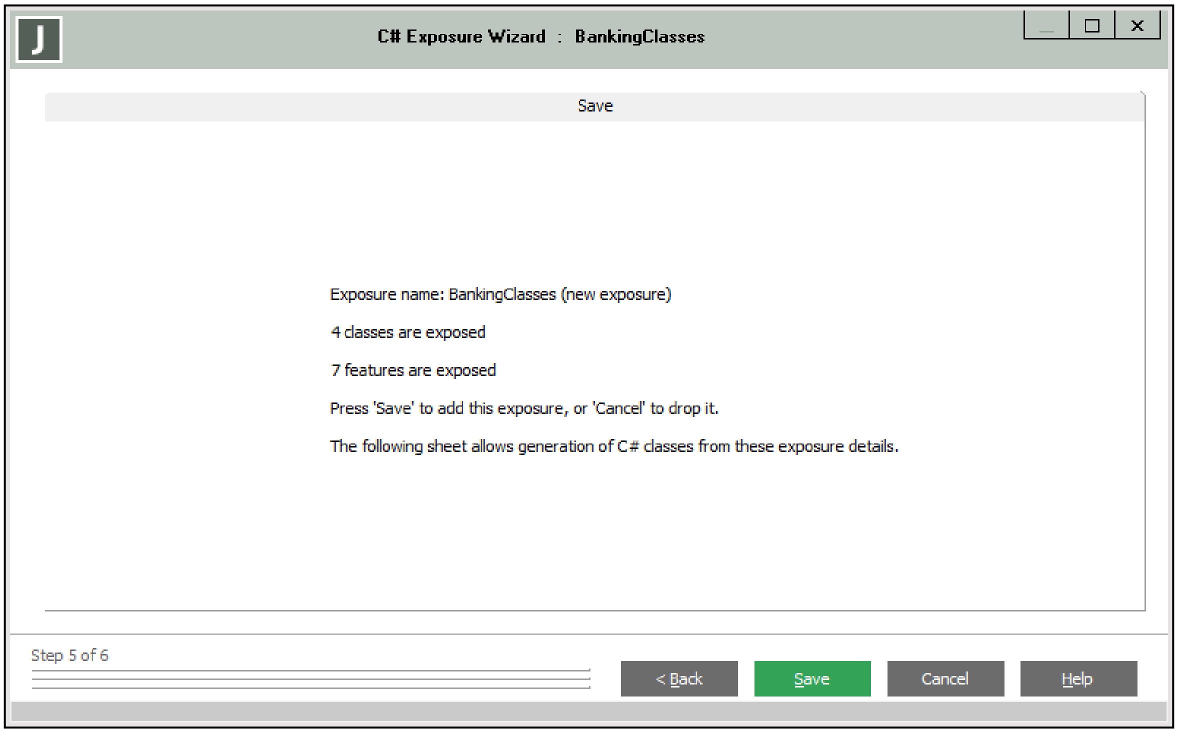
Jade Feature	C# Feature Name	C# Type
allCustomers	AllCustomers	CustomerByLastNameDic
myBank	MyBank	Bank
setPropsOnCreate	SetPropsOnCreate	void
address	Address	String
firstNames	FirstNames	String
lastName	LastName	String
number	Number	Int32

[✓] Modify All Classes

By default:

- C# class names are the same as those of the corresponding JADE classes
- C# property and method names are derived from the corresponding JADE names with the first letter changed to uppercase

8. The Save step of the wizard summarizes the exposure definition to be saved.



Exposure name: BankingClasses (new exposure)

4 classes are exposed

7 features are exposed

Press 'Save' to add this exposure, or 'Cancel' to drop it.

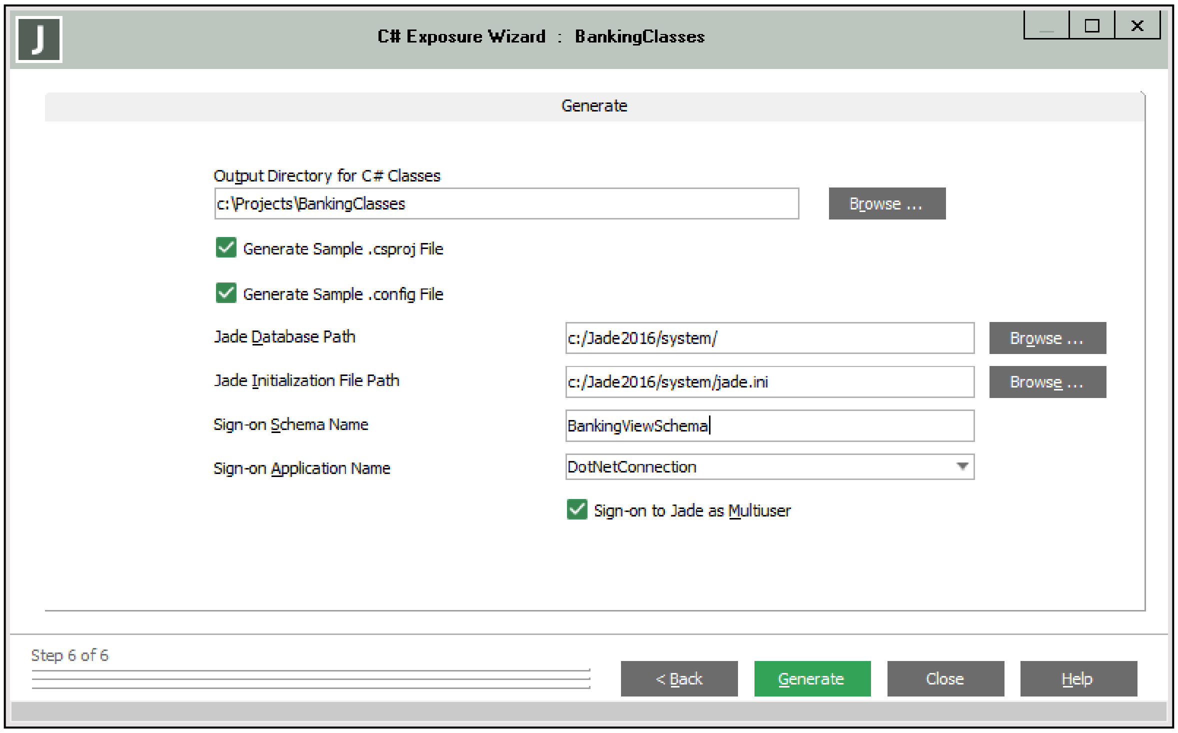
The following sheet allows generation of C# classes from these exposure details.

9. On the Generate step of the wizard, enter **C:\Projects\BankingClasses** as the directory where the C# project file and class files will be created.

   Check the option to create a sample C# project file. In addition, check the option to create an application configuration file using the following information specified on the form.

   - JADE database path
   - JADE initialization file location
   - Sign-on schema name
   - Sign-on application name, for which you should select **DotNetConnection**
   - Mode (multiuser or single user) in which the JADE database should be open when the C# application attempts to connect

Note that this application configuration information is similar to that provided in the shortcut for a standard JADE client.



Output Directory for C# Classes: c:\Projects\BankingClasses

☑ Generate Sample .csproj File

☑ Generate Sample .config File

Jade Database Path	c:/Jade2016/system/
Jade Initialization File Path	c:/Jade2016/system/jade.ini
Sign-on Schema Name	BankingViewSchema
Sign-on Application Name	DotNetConnection

☑ Sign-on to Jade as Multiuser

10. Click the **Generate** file to generate the files when you build a C# project in the next section.

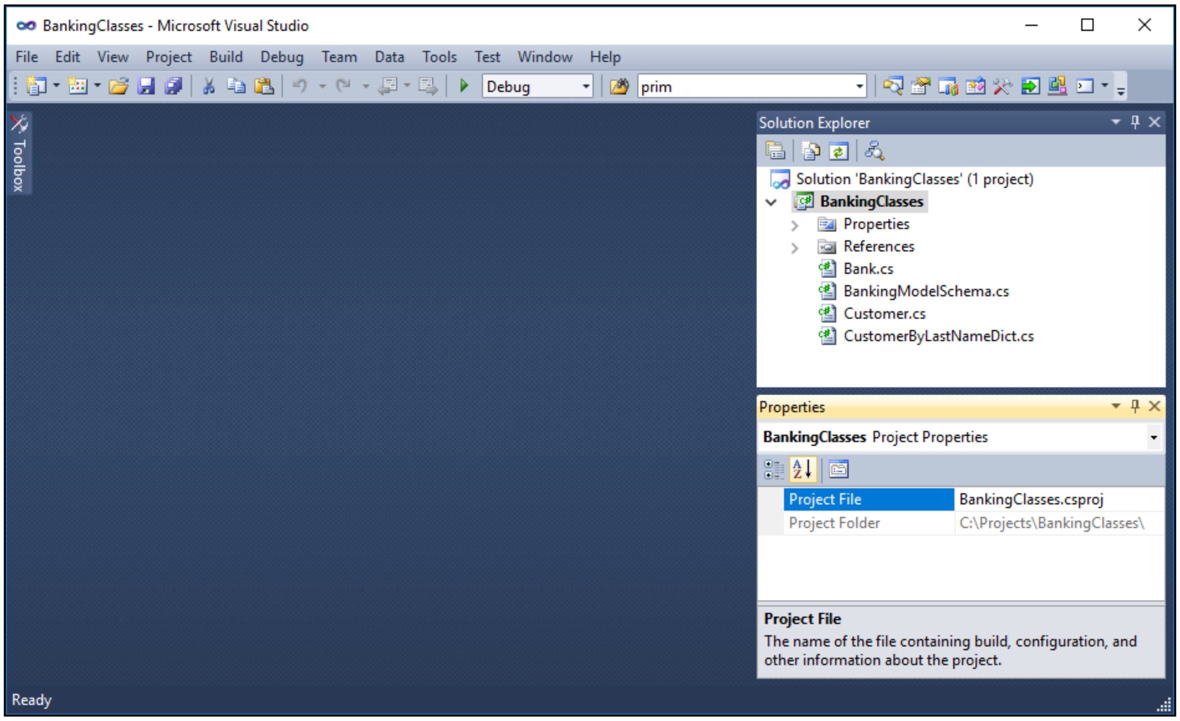
Building the Class Exposure Project in Visual Studio

In the final step of the C# Exposure wizard, a number of C# class files and other files were generated, including the project file **BankingClasses.csproj**.

The instructions in this section enable you to build a class library DLL for the exposed classes from the JADE banking system.

The DLL can be referenced by other .NET projects, enabling them to create and access instances of the **Customer** class.

1. Open the **BankingClasses.csproj** file in Visual Studio.

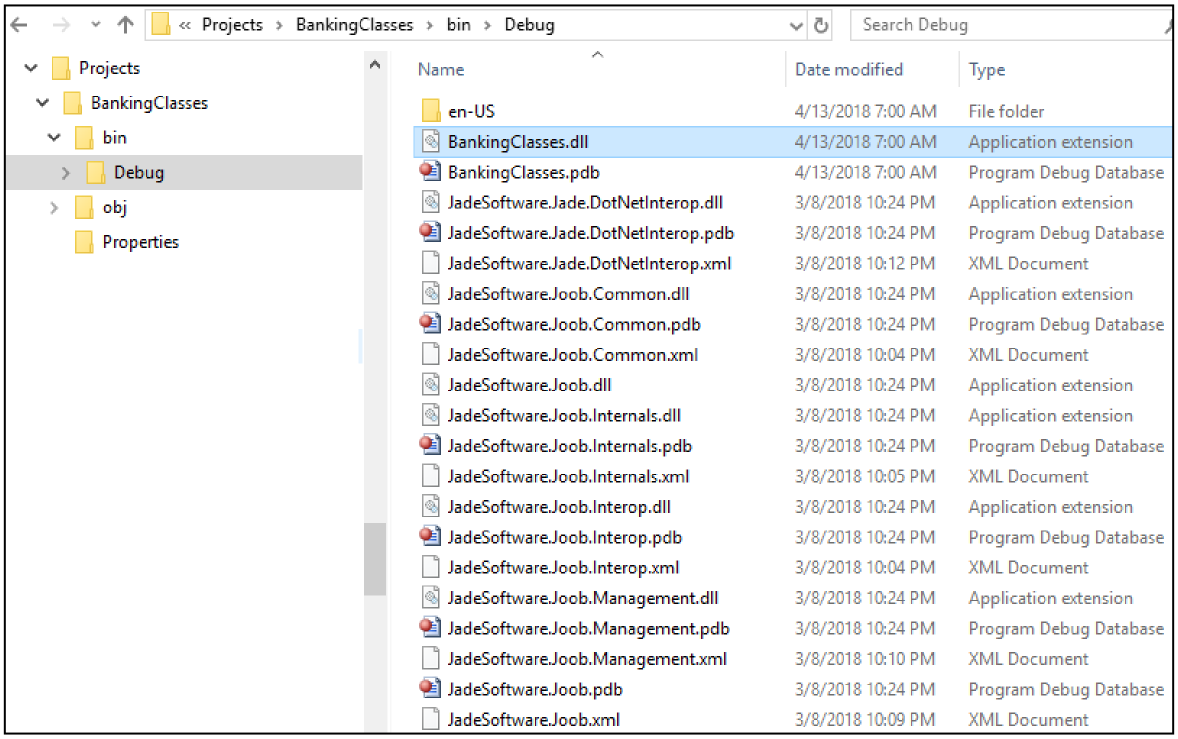

BankingClasses - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Test Window Help
Debug
prim
Toolbox
Solution Explorer
Solution 'BankingClasses' (1 project)
BankingClasses
Properties
References
Bank.cs
BankingModelSchema.cs
Customer.cs
CustomerByLastNameDict.cs
Properties
BankingClasses Project Properties
Ready

Project File	BankingClasses.csproj
Project Folder	C:\Projects\BankingClasses\

Project File
The name of the file containing build, configuration, and other information about the project.

The classes exposed in the C# Exposure wizard are visible in the Solution Explorer.

2. Build the project by selecting the **Build BankingClasses** command from the Visual Studio Build menu.



Name	Date modified	Type
en-US	4/13/2018 7:00 AM	File folder
BankingClasses.dll	4/13/2018 7:00 AM	Application extension
BankingClasses.pdb	4/13/2018 7:00 AM	Program Debug Database
JadeSoftware.Jade.DotNetInterop.dll	3/8/2018 10:24 PM	Application extension
JadeSoftware.Jade.DotNetInterop.pdb	3/8/2018 10:24 PM	Program Debug Database
JadeSoftware.Jade.DotNetInterop.xml	3/8/2018 10:12 PM	XML Document
JadeSoftware.Joob.Common.dll	3/8/2018 10:24 PM	Application extension
JadeSoftware.Joob.Common.pdb	3/8/2018 10:24 PM	Program Debug Database
JadeSoftware.Joob.Common.xml	3/8/2018 10:04 PM	XML Document
JadeSoftware.Joob.dll	3/8/2018 10:24 PM	Application extension
JadeSoftware.Joob.Internals.dll	3/8/2018 10:24 PM	Application extension
JadeSoftware.Joob.Internals.pdb	3/8/2018 10:24 PM	Program Debug Database
JadeSoftware.Joob.Internals.xml	3/8/2018 10:05 PM	XML Document
JadeSoftware.Joob.Interop.dll	3/8/2018 10:24 PM	Application extension
JadeSoftware.Joob.Interop.pdb	3/8/2018 10:24 PM	Program Debug Database
JadeSoftware.Joob.Interop.xml	3/8/2018 10:04 PM	XML Document
JadeSoftware.Joob.Management.dll	3/8/2018 10:24 PM	Application extension
JadeSoftware.Joob.Management.pdb	3/8/2018 10:24 PM	Program Debug Database
JadeSoftware.Joob.Management.xml	3/8/2018 10:10 PM	XML Document
JadeSoftware.Joob.pdb	3/8/2018 10:24 PM	Program Debug Database
JadeSoftware.Joob.xml	3/8/2018 10:09 PM	XML Document

The **BankingClasses.dll** library is created in the **bin\Debug** directory of the project. The DLLs from the JADE .NET framework are copied into the same location.

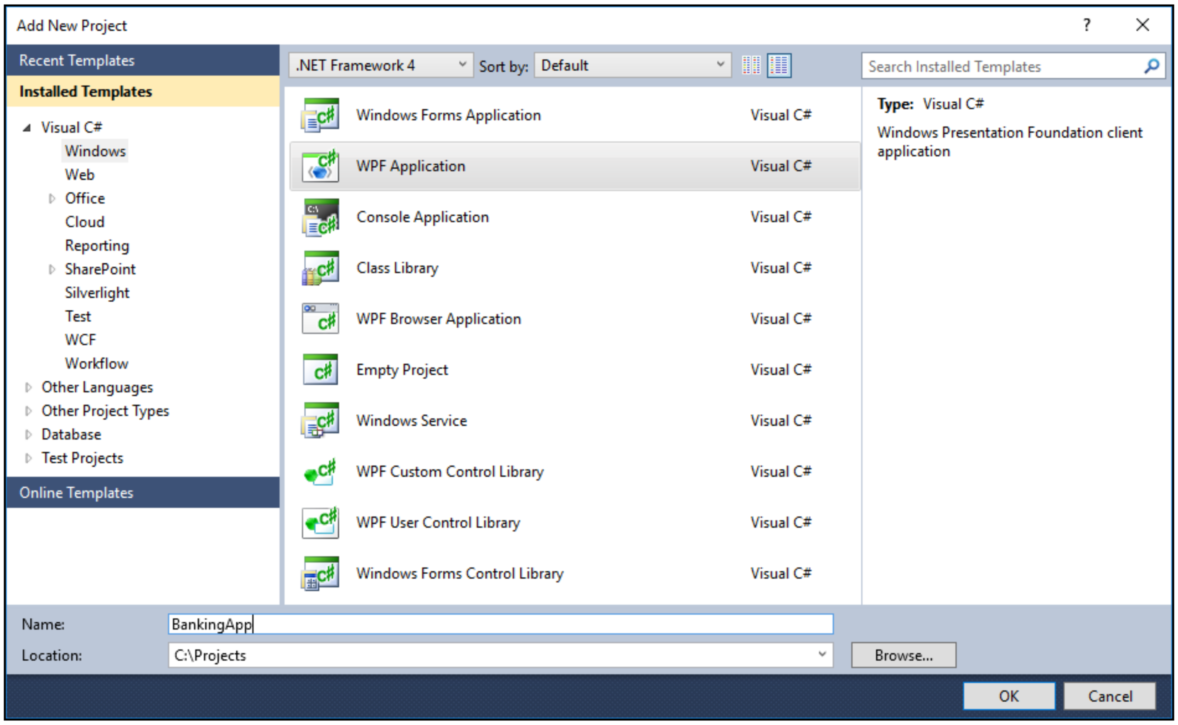
3. If prompted to do so, click the **Save All** icon and then enter **C:\Projects\BankingSystem.sln** when prompted for the name and path of the solution file.

Adding a WPF Application

The following instructions create a WPF project called **BankingApp** to the solution and set the application as the default one to be run for the solution (currently the **BankingClasses** project is the default project). Finally, the project properties are changed to build a 64-bit application.

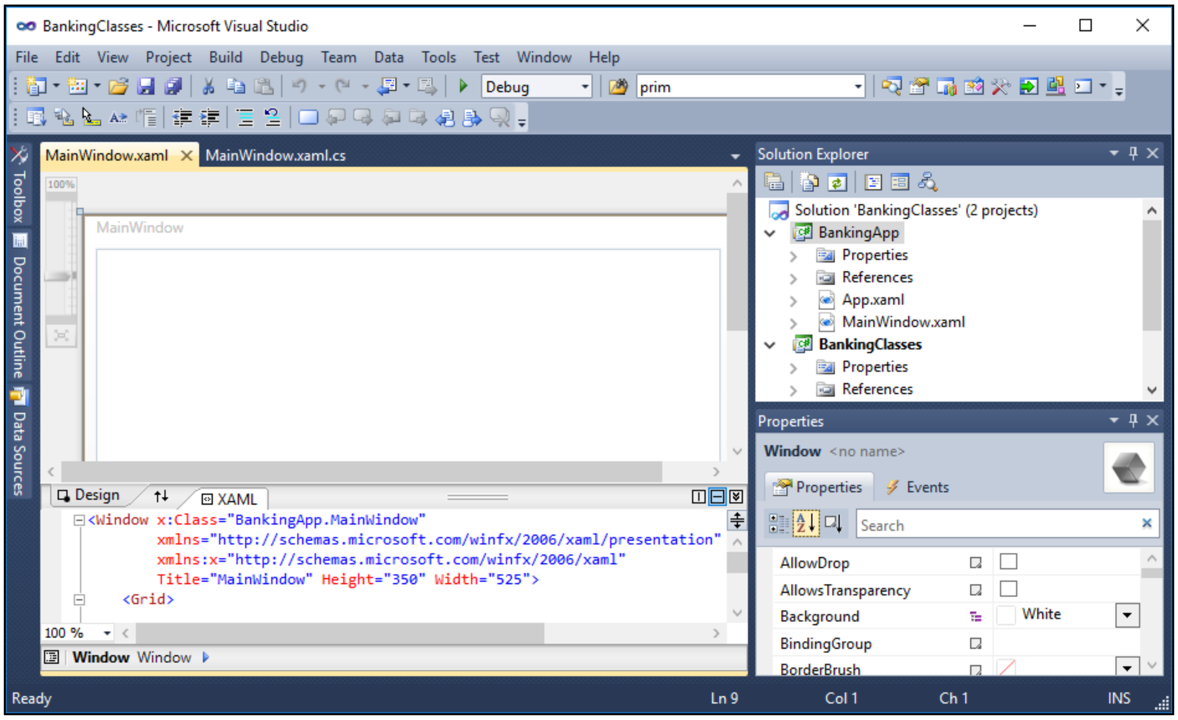
1. Right-click on the **BankingSystem** solution and then select the **New Project** command from the Add submenu, to add a WPF project.

Complete the dialog, as shown in the following image.


Add New Project
Recent Templates
Installed Templates
Visual C#
Windows
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Test
WCF
Workflow
Other Languages
Other Project Types
Database
Test Projects
Online Templates
.NET Framework 4
Sort by: Default
Search Installed Templates
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
Class Library Visual C#
WPF Browser Application Visual C#
Empty Project Visual C#
Windows Service Visual C#
WPF Custom Control Library Visual C#
WPF User Control Library Visual C#
Windows Forms Control Library Visual C#
Type: Visual C#
Windows Presentation Foundation client application
Browse...
OK
Cancel

Name: BankingApp
Location: C:\Projects

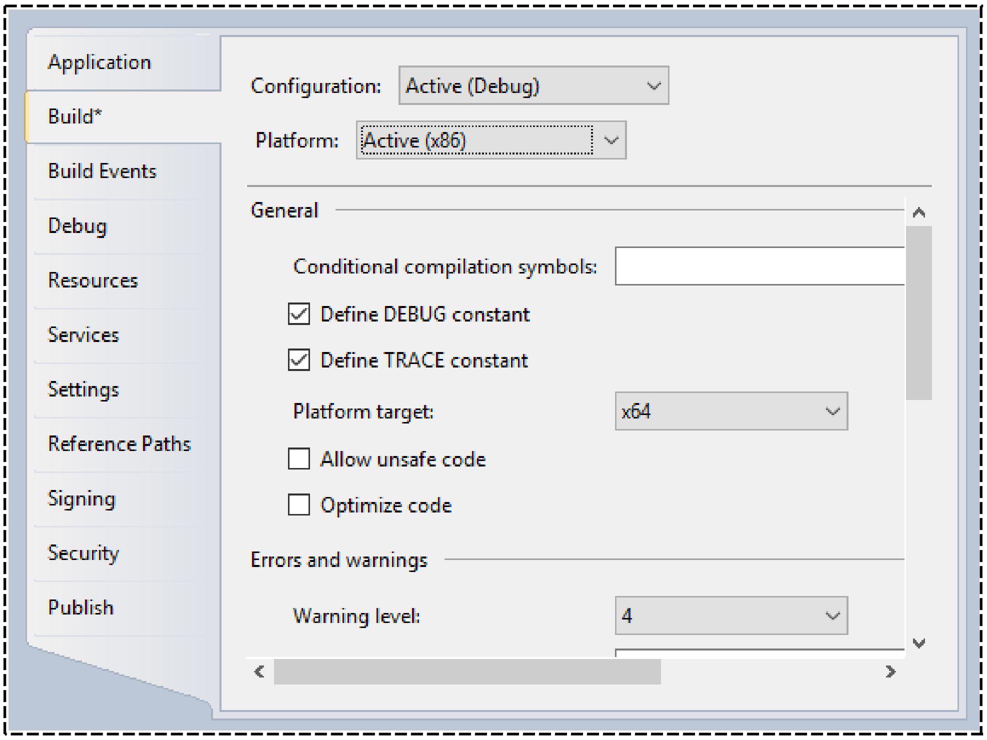
2. The Solution Explorer shows the **BankingSystem** solution with two projects: **BankingApp** and **BankingClasses**.



```
<Window x:Class="BankingApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
```

3. Right-click on the **BankingApp** project and then select **Set as Startup Project** from the menu. This causes the application to be run when debugging is started.

4. Right-click on the **BankingApp** project and then select **Properties** from the menu. On the **Build** tab, change the **Platform target** to be **x64**.



Configuration: Active (Debug)

Platform: Active (x86)

Conditional compilation symbols:

☑ Define DEBUG constant

☑ Define TRACE constant

Platform target: x64

☐ Allow unsafe code

☐ Optimize code

Warning level: 4

Adding References

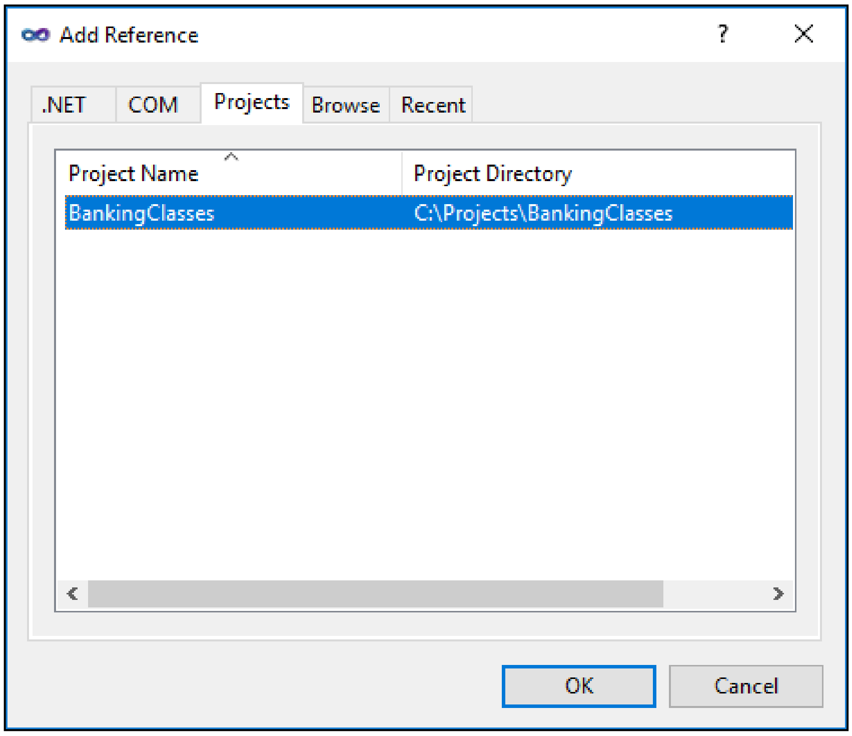
For the **BankingApp** project to work with the exposed classes in the **BankingClasses** project, a reference must be added to the **BankingClass.dll** that was built earlier. In addition, references to the JADE .NET framework DLLs must be added.

To simplify your application coding that uses classes from the referenced DLLs, **using** directives are added to the start of the **MainWindow.xaml.cs** file.

The following instructions add the required references and **using** directives.

1. Right-click on the **BankingApp** project then and then select the **Add Reference** command.

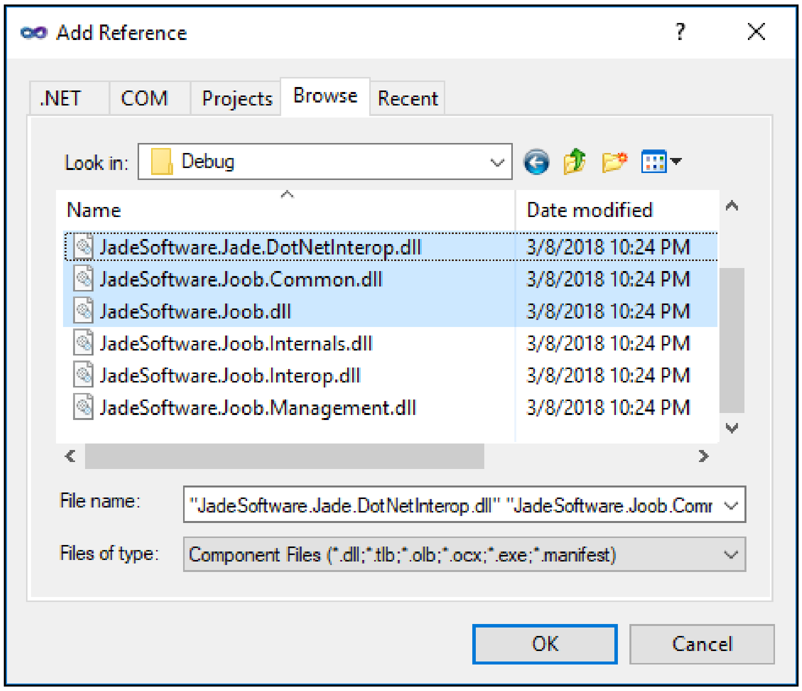
   On the **Projects** sheet of the Add Reference dialog, select **BankingClasses** from the solution assemblies.



Project Name	Project Directory
BankingClasses	C:\Projects\BankingClasses

2. Using the Add Reference dialog again, add references to the following JADE .NET framework DLLs by browsing for their location.

   - JadeSoftware.Joob.dll
   - JadeSoftware.Joob.Common.dll
   - JadeSoftware.Jade.DotNetInterop.dll



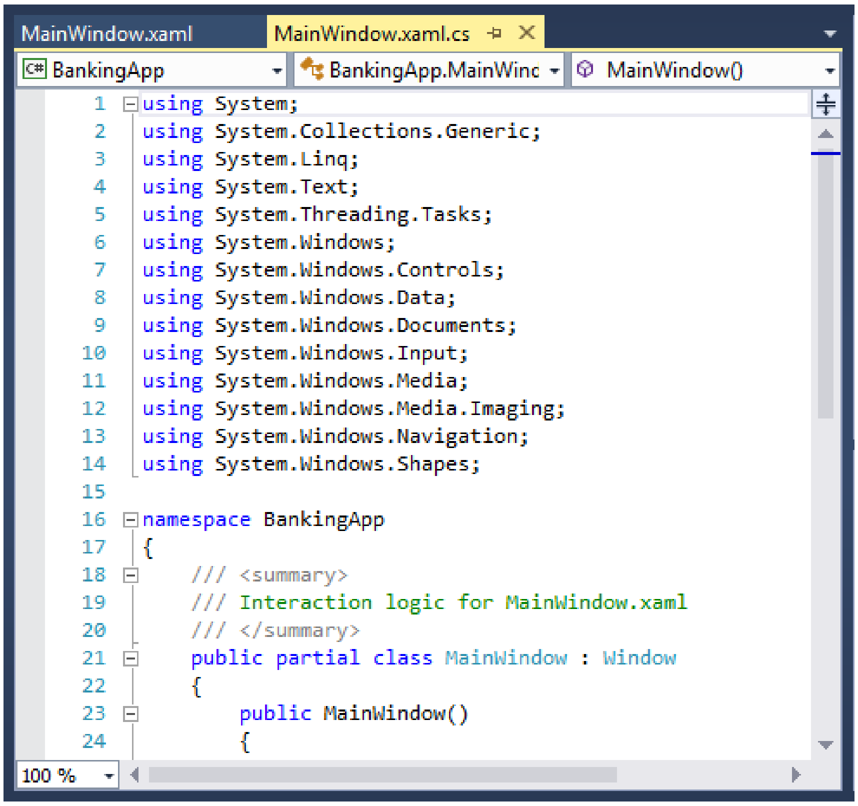
Look in: Debug

Name	Date modified
JadeSoftware.Jade.DotNetInterop.dll	3/8/2018 10:24 PM
JadeSoftware.Joob.Common.dll	3/8/2018 10:24 PM
JadeSoftware.Joob.dll	3/8/2018 10:24 PM
JadeSoftware.Joob.Internals.dll	3/8/2018 10:24 PM
JadeSoftware.Joob.Interop.dll	3/8/2018 10:24 PM
JadeSoftware.Joob.Management.dll	3/8/2018 10:24 PM

File name: "JadeSoftware.Jade.DotNetInterop.dll" "JadeSoftware.Joob.Comr

Files of type: Component Files (*.dll;*.tlb;*.olb;*.ocx;*.exe;*.manifest)

3. Select the tab for the **MainWindow.xaml.cs** code file.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace BankingApp
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
```

4. Add the following **using** directives for namespaces to your code.

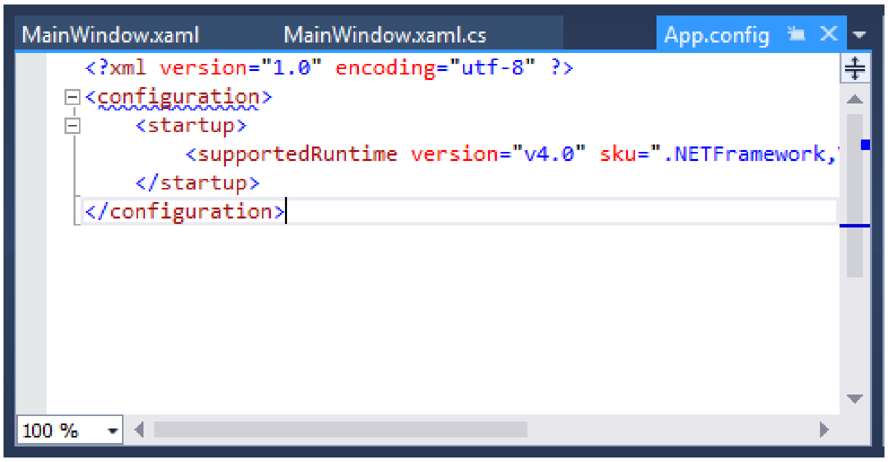
```
using BankingClasses;
using JadeSoftware.Joob;
using JadeSoftware.Joob.Client;
using JadeSoftware.Joob.Exceptions;
using JadeSoftware.Jade.DotNetInterop;
```

Coding the Application Configuration File

The connection string information in the application configuration is used by the .NET application to connect to the database server. It contains the same kind of information you would see in a JADE client shortcut.

An example application configuration file called **BankingClasses.exe.config** was generated by the C# Exposure wizard, so if your information provided to the wizard was correct, you can use the contents of this file without modification.

1. Open the **App.config** file for the **BankingApp** project.



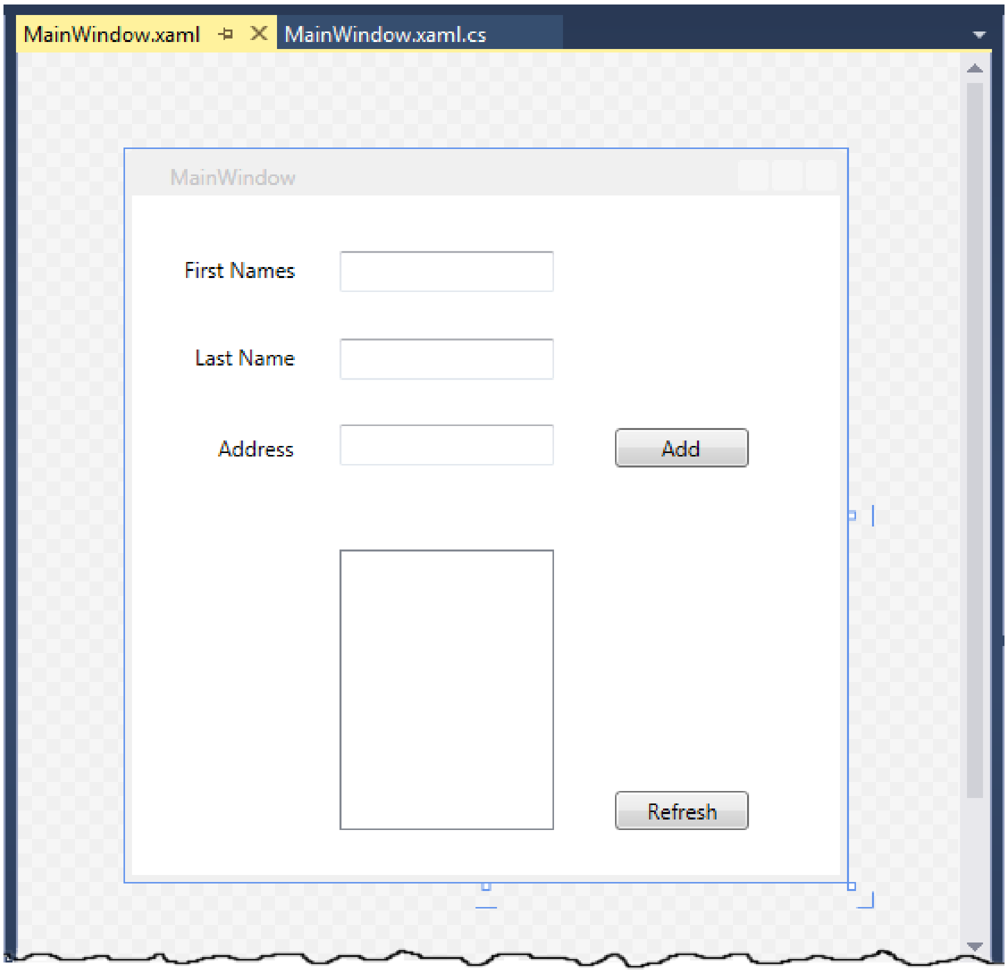
```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,'
    </startup>
</configuration>
```

2. Overwrite the XML in this file with the contents of the **BankingClasses.exe.config** file.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="joob"
             type="JadeSoftware.Joob.Configuration.JoobConfigurationSection,
                   JadeSoftware.Joob" />
  </configSections>

  <connectionStrings>
    <add name="myDefault" providerName="JadeSoftware.Joob.JadeConnection"
         connectionString="DataSource=C:/Jade7009/system/;
                           ConfigFile=C:/Jade7009/system/jade.ini;
                           SingleUser=False;
                           Schema=BankingViewSchema;
                           Application=DotNetConnection;
                           IntegratedSecurity=true"></add>
  </connectionStrings>

  <joob defaultConnection="myDefault">
    <installation directory="C:\Jade7009\bin" />
  </joob>

  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/>
  </startup>
</configuration>
```

Designing the Form

The instructions in this section enable you to design a form to add a customer and to display a list of customer names. The completed form layout is as follows.



1. Add three **Label** controls and set the **Content** property to **First Names**, **Last Name**, and **Address**.
2. Add three **TextBox** controls next to the labels and then set the **Name** to **txtFirstNames**, **txtLastName**, and **txtAddress**. In addition, set the **Text** property of the text boxes to be an empty string.
3. Add a **ListBox** control and then set its **Name** to **lstCustomers**.
4. Add a **Button** control with the name **btnAdd** and then set its **Content** property to **Add**.
5. Add a **Button** control with the name **btnRefresh** and then set its **Content** property to **Refresh**.

Coding the Form Constructor and Destructor

Before you can work with **Customer** objects, you need to connect to the database. This is provided by a **JoobContext** object.

The simplest **JoobContext** constructor is the default no-parameters constructor, which uses the **defaultConnection** element from the application configuration file.

To establish a connection, simply create a **JoobContext** object as follows.

```
JoobContext context = new JoobContext();
```

When a connection is no longer needed, dispose of the **JoobContext** object.

```
context.Dispose();
```

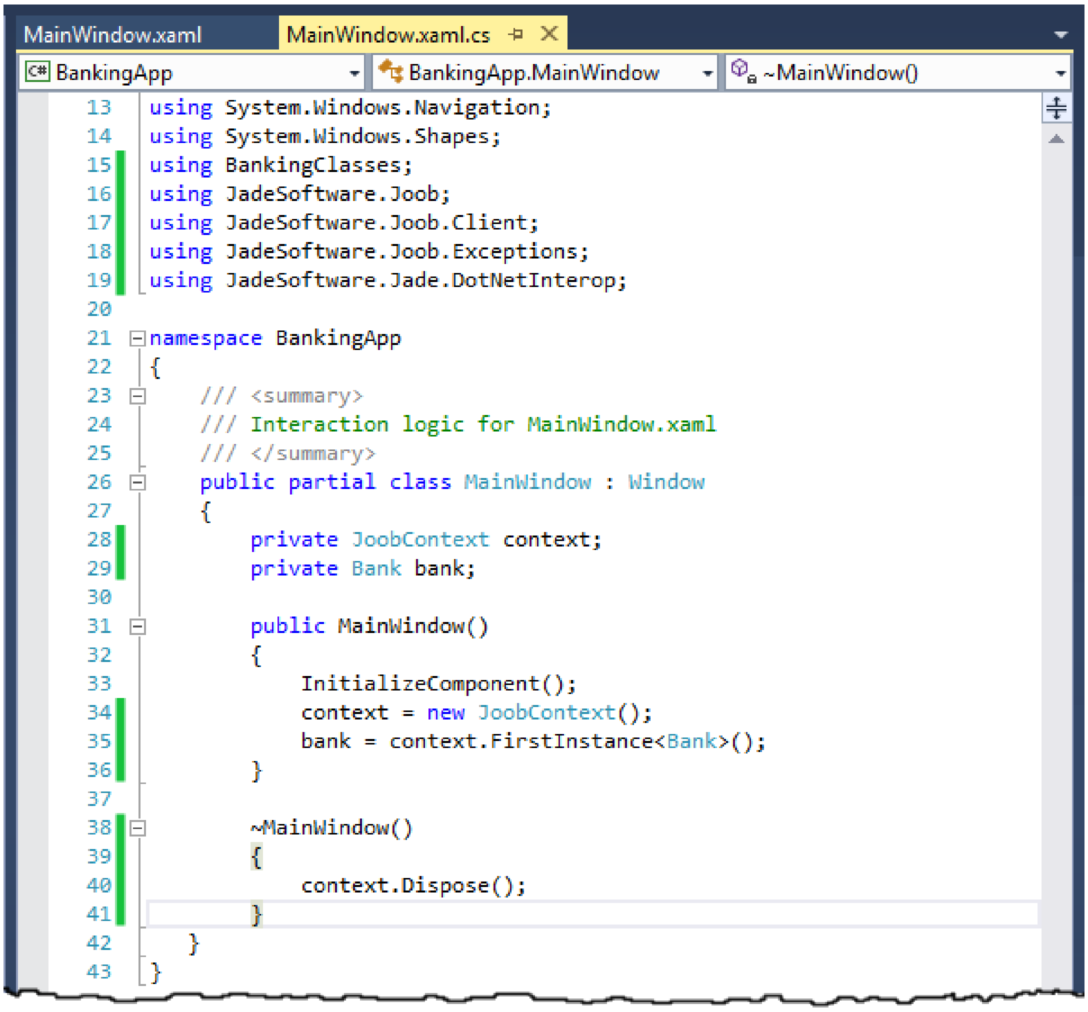
If a form utilizes the connection for the entire time it is shown, the form should have a reference to a context object. This reference is available to all code in the form. You can create the context object when the form is opened and dispose of this object when the form is closed.

If the form requires an occasional connection to the JADE database only (perhaps only if a specific button on the form is clicked), you can establish a temporary connection by creating a context object in a **using** block, as follows.

```
using (JoobContext context = new JoobContext())
{
    // access the database
}
```

Another object that is used throughout the application is the root object. The **JoobContext** object provides a way of accessing this object using a template method that can return the first instance of any class. The first (and only) instance of the **Bank** class is the root object.

```
bank = context.FirstInstance<Root>();
```


MainWindow.xaml | MainWindow.xaml.cs

BankingApp | BankingApp.MainWindow | ~MainWindow()

```
using System.Windows.Navigation;
using System.Windows.Shapes;
using BankingClasses;
using JadeSoftware.Joob;
using JadeSoftware.Joob.Client;
using JadeSoftware.Joob.Exceptions;
using JadeSoftware.Jade.DotNetInterop;

namespace BankingApp
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private JoobContext context;
        private Bank bank;

        public MainWindow()
        {
            InitializeComponent();
            context = new JoobContext();
            bank = context.FirstInstance<Bank>();
        }

        ~MainWindow()
        {
            context.Dispose();
        }
    }
}
```

In the following instructions, a context object is created for the life of the form and a reference to the root object established.

1. In the **MainWindow.xaml.cs** code file, add private member variables called **context** and **bank** to the **MainWindow** class for the context object and the root object, respectively.

```
public partial class MainWindow: Window
{
    private JoobContext context;
    private Bank bank;
```

2. In the form constructor, initialize the **context** and **bank** references.

```
public MainWindow()
{
    InitializeComponent();
```

```
        context = new JoobContext();
        bank = context.FirstInstance<Bank>();
    }
```

3. Add a form destructor to dispose the **JoobContext** object.

```
    ~MainWindow()
    {
        context.Dispose();
    }
```

Listing Customers

The **Bank** root object's collection of **Customer** objects is to be displayed in the **lstCustomers** list box. For simplicity, only the surname of the customer is displayed.

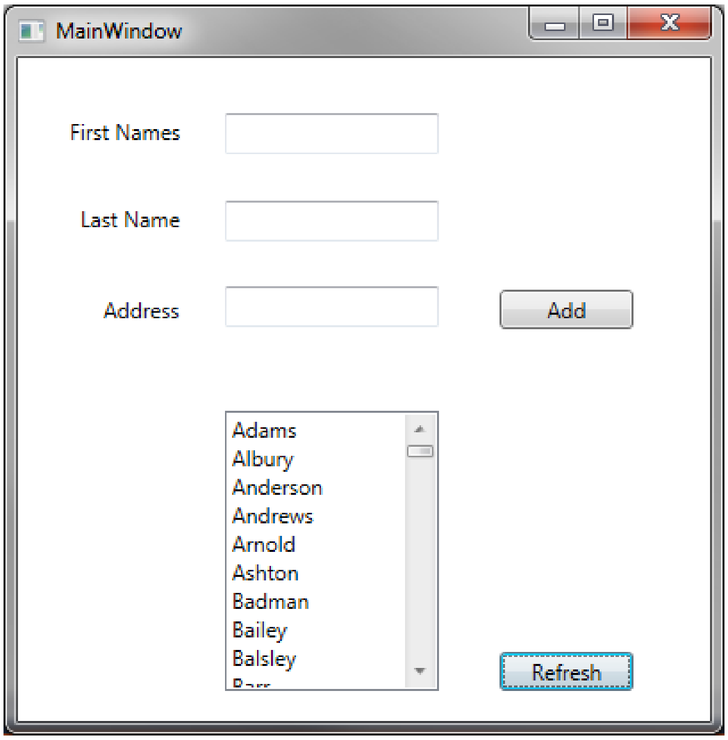
1. In the **MainWindow.xaml** form, double-click the **Refresh** button. This adds a **click** event method in the **MainWindow.xaml.cs** code file.

2. Add the following code to the **btnRefresh_Click** method.

```
    private void btnRefresh_Click(object sender, RoutedEventArgs e)
    {
       lstCustomers.ItemsSource = bank.AllCustomers;
       lstCustomers.DisplayMemberPath = "LastName";
    }
```

3. Run the **BankingApp** application and then click the **Refresh** button.

   If you have created **Customer** objects previously as part of the JADE developer's course, they are displayed.



Adding a Customer

The **JoobContext** object has a method for beginning a transaction. The method returns a transaction object with methods for committing or rolling back a transaction. The transaction object must be disposed of at the end of the transaction.

The simplest syntax for carrying out a transaction uses a **using** statement that implicitly calls the **Dispose** method on the transaction object at the end of the block.

The **SetPropertiesOnCreate** method, which was exposed by the C# Exposure wizard, sets properties of the **Customer** object from values entered in the text boxes on the form.

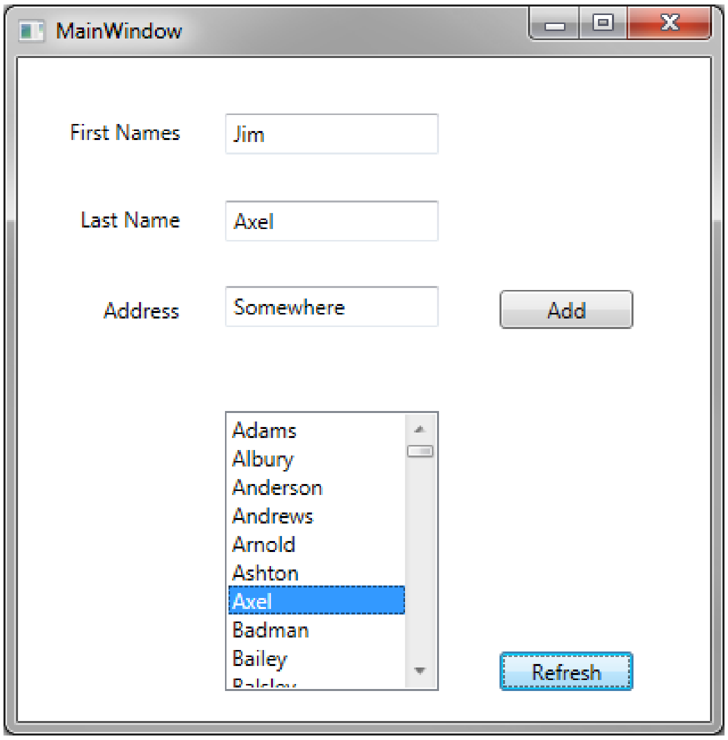
If the statement calling the **Commit** method is omitted or not executed because an exception is thrown, the transaction is automatically rolled back. With a **using** statement, you do not have to explicitly call the **Rollback** method.

1. In the **MainWindow.xaml** form, double-click the **Add** button. This adds a **click** event method in the **MainWindow.xaml.cs** code file.

2. Add the following code to the **btnAdd_Click** method.

```
private void btnAdd_Click(object sender, RoutedEventArgs e)
{
    using (var tran = context.BeginTransaction())
    {
        Customer cust = new Customer();
        cust.SetPropertiesOnCreate(txtAddress.Text,
                                   txtFirstNames.Text,
                                   txtLastName.Text);
        tran.Commit();
    }
}
```

3. Run the **BankingApp** application and the enter data for a new customer. Click the **Add** button to add the customer and then click the **Refresh** button to display the new customer.



Using Notifications

The **JoobContext** class enables you to register for notifications; that is, subscribe to events. The **RegisterClassNotificationHandler** method, which registers for notifications for a specific event that happens to any instance of the specified class, has the following parameters.

- Class (when the event happens to an instance of the class, a notification is sent)
- Event type (represented by an integer)
- Event handler method (called when the event occurs)

There is also an **UnregisterClassNotificationHandler** method.

One use of notifications is to automatically refresh the information displayed in an application when the underlying information changes. The instructions in this section are intended to make the **Refresh** button redundant.

1. In the **MainWindow.xaml.cs** code file, add a private **refresh** method with the same code as the **btnRefresh_ Click** method

```
private void refresh()
{
    lstCustomers.ItemsSource = bank.AllCustomers;
    lstCustomers.DisplayMemberPath = "LastName";
}
```

2. In the form constructor, register to be notified when **Customer** objects are created. The **customerCreatedNotificationHandler** method, which you will code shortly, is the event handler for the notification.

```
public MainWindow()
{
    InitializeComponent();
    context = new JoobContext();
    bank = context.FirstInstance<Bank>();
    context.RegisterClassNotificationHandler(typeof(Customer),
                                   NotificationEventConstants.SystemCreate,
                                   customerCreatedNoteHandler);
```

3. Add a **customerCreatedNoteHandler** notification event handler method.

```
private void customerCreatedNoteHandler(object sender, NotificationEventArgs e)
{
    MessageBox.Show("Customer added");
    // cannot run refresh() yet because not on GUI thread
}
```

4. Run the **BankingApp** application form and enter data for a new customer. Click the **Add** button to add the customer. A message box stating that the customer has been added is then displayed.

Notifications are delivered on a separate thread from the GUI thread on which the application runs. The notifications thread can display a message box, as you have seen, but cannot see the list box or other controls on the form. Consequently the **refresh** method cannot be executed directly from this thread.

Invoking on the GUI Thread

As explained in the previous section, the notification is received by a background thread and not by the GUI thread on which the application is running. Although the **refresh** method cannot be run directly it can be invoked using a delegate.

1. In the **MainWindow.xaml.cs** code file, define a delegate type for refreshing the display that takes no parameters called **RefreshDelegate** and declare a member of this type.

```
public partial class MainWindow: Window
{
    private JoobContext context;
    private Bank bank;
    delegate void RefreshDelegate();
    RefreshDelegate refreshDelegate;
```

2. In the form constructor, point the **refreshDelegate** member to the **refresh** method.

```
public MainWindow()
{
    InitializeComponent();
    context = new JoobContext();
    bank = context.FirstInstance<Bank>();
    context.RegisterClassNotificationHandler(typeof(Customer),
                                   NotificationEventConstants.SystemCreate,
                                   customerCreatedNoteHandler);
    refreshDelegate = new RefreshDelegate(refresh);
```

3. In the **customerCreatedNoteHandler** notification event handler method, use the **Invoke** method to marshal the **refresh** method to the GUI thread.

```
private void customerCreatedNoteHandler(object sender, NotificationEventArgs e)
{
    MessageBox.Show("Customer added");
    Dispatcher.Invoke(refreshDelegate);
}
```

4. Run the **BankingApp** application and then enter data for a new customer. Click the **Add** button to add the customer. After the message box is closed, the list of customers is refreshed automatically.

Chapter 4 Erewhon .NET Example

This chapter covers the following topics.

Overview

This chapter describes the JADE .NET class exposure example for the JADE Erewhon demonstration system, which is an Internet-enabled online purchasing and tendering application. You can download the **JADE-Erewhon** example system from https://github.com/jadesoftwarenz.

The Erewhon .NET example consists of an exposure of classes from the Erewhon system that is used by a WPF application to access the JADE database. The JADE .NET class library functions can be executed in the WPF application to execute a number of actions.

- Display a list of **Agent** objects
- Create, update, and delete a **Client** object
- Register and unregister for notifications on system events
- Register and unregister for notifications on user events
- Lock and unlock a **Client** object
- Arm exception handlers and handle an exception
- Run LINQ queries
- Use multithreading

The code in the WPF application is for demonstration purposes only and is not intended for production systems.

Example .NET Files

The file **ErewhonDotNetExample.zip** file contains the following directories.

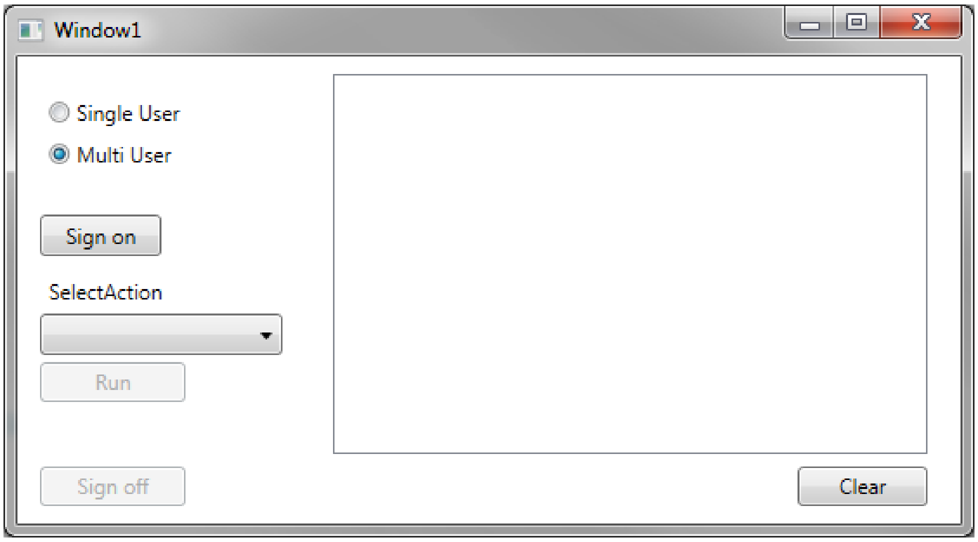
- ErewhonExposure
- WpfErewhonApp

The **ErewhonExposure** directory contains the files created by running the Exposure wizard from the JADE development environment. This directory was specified as the output directory of the wizard. If you use the wizard to expose all classes in the **ErewhonInvestmentModelSchema** schema, you would get a matching set of files.

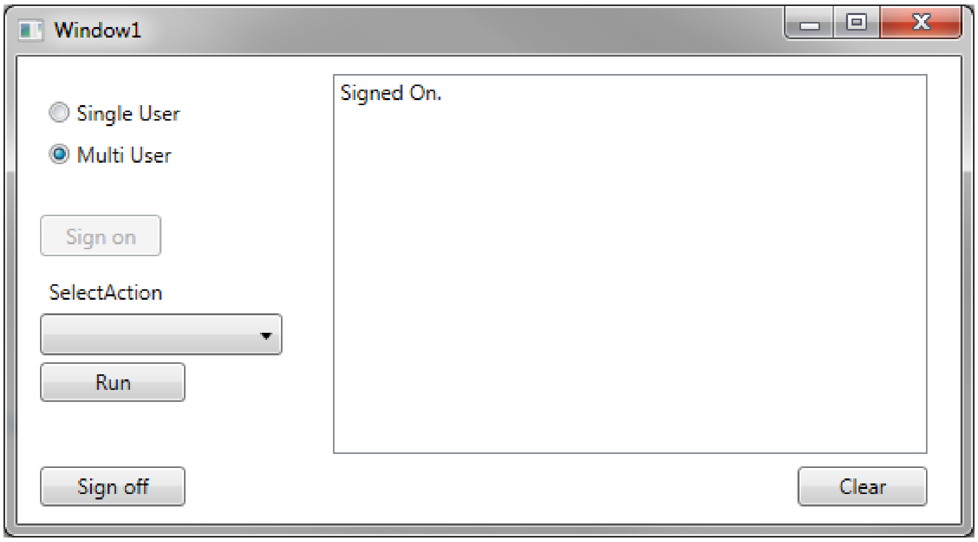
The **WpfErewhonApp** directory contains the files and directories for a WPF solution built in Visual Studio. The example application code has been added to this solution.

Running the WpfErewhonApp Application

When you run the **WpfErewhonApp** application, the following form is displayed.

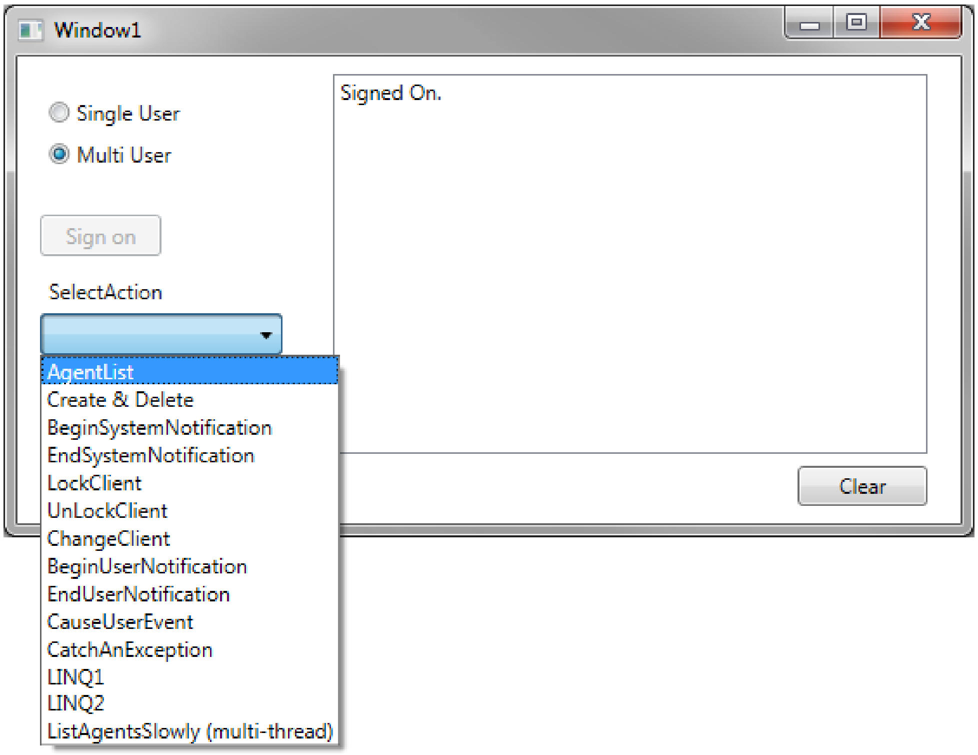


You must first sign on by selecting **Single User** or **Multi User**, depending on how the database is opened, and then clicking the **Sign on** button. A message is displayed in the logging window.



Signed On.

The **SelectAction** combo box displays a list of functions that you can run.



Signed On.

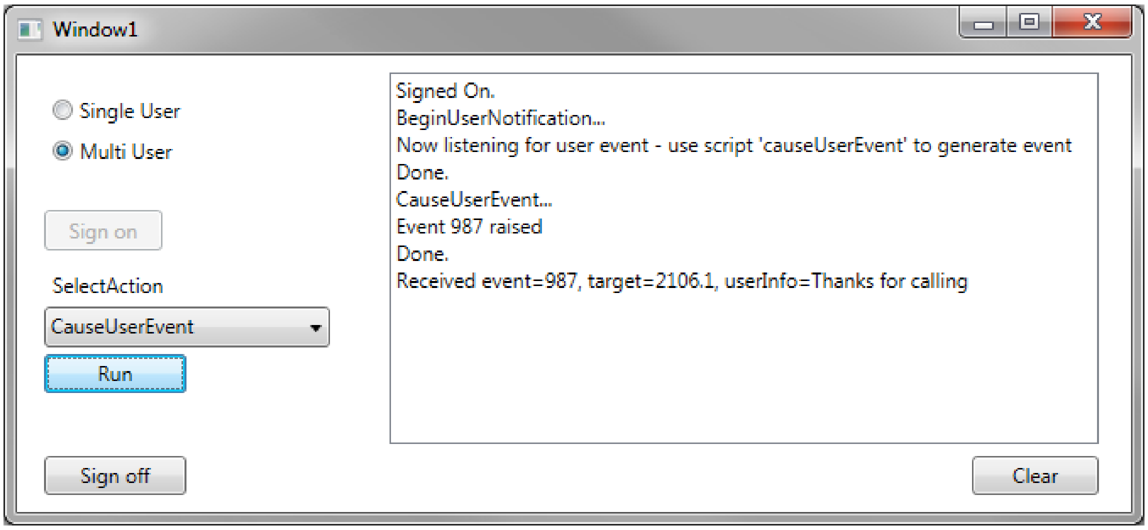
After selecting an action, click the **Run** button. The status and any output generated by that action is recorded in the logging window.

Notifications

To demonstrate user notifications, a number of actions must be run in the correct order.

1. Sign on to the **WpfErewhonApp** application.
2. Run the **BeginUserNotification** action to register for user notifications.

3. Run the **CauseUserEvent** action. A message is displayed in the logging window, indicating a user event has been received.



```
Signed On.
BeginUserNotification...
Now listening for user event - use script 'causeUserEvent' to generate event
Done.
CauseUserEvent...
Event 987 raised
Done.
Received event=987, target=2106.1, userInfo=Thanks for calling
```

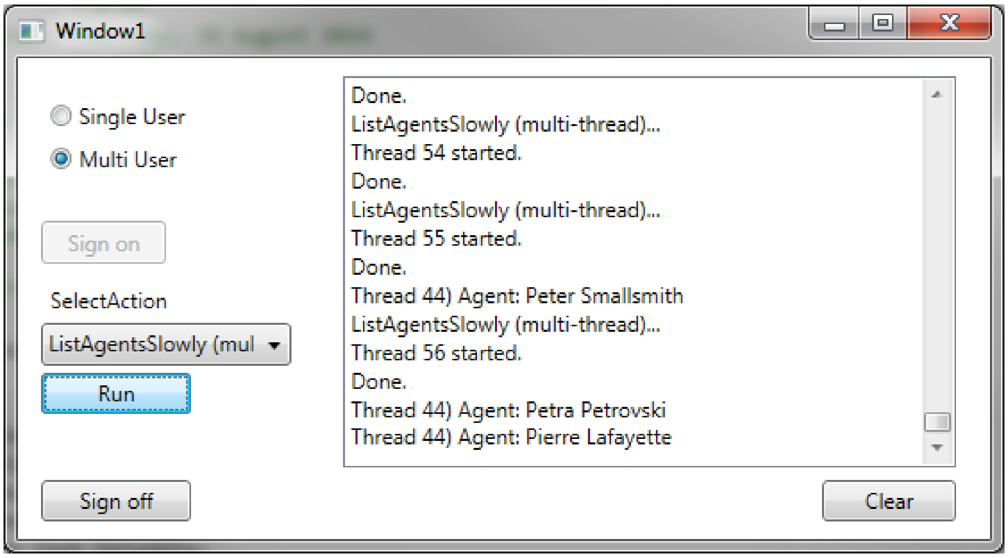
4. Run the **EndUserNotification** action to stop receiving user notifications.

5. Sign off from the **WpfErewhonApp** application.

Examine the application code for comments that provide additional explanation.

Multiprocessing

Each time you run the **ListAgentsSlowly** action, it is run on a new thread. If you click the **Run** button repeatedly, multiple threads are started, each executing the **ListAgentsSlowly** method.



```
Done.
ListAgentsSlowly (multi-thread)...
Thread 54 started.
Done.
ListAgentsSlowly (multi-thread)...
Thread 55 started.
Done.
Thread 44) Agent: Peter Smallsmith
ListAgentsSlowly (multi-thread)...
Thread 56 started.
Done.
Thread 44) Agent: Petra Petrovski
Thread 44) Agent: Pierre Lafayette
```

Running the **ListAgentsSlowly** action repeatedly can result in *Busy* messages being displayed in the logging window.

Building Your Own Application

The following instructions outline how you can build your own WPF application to connect to the JADE database.

1. Create a new WPF project.
2. Build the project and solution.
3. Add the existing project created by the Exposure wizard, **ErewhonExposure**, to this solution.
4. In the WPF project you created in the first step, add references to:
   - JadeSoftware.Jade.DotNetInterop.dll
   - JadeSoftware.Joob.dll
   - JadeSoftware.Joob.Common.dll
   - ErewhonExposure.dll (the exposure project added to the solution)
5. Ensure that the target platform is set to **x64** (64-bit) on the **Build** tab of the Properties sheet for your WPF application.

Chapter 5 Using the JADE .NET Framework

This chapter covers the following topics.

Overview

The JADE .NET framework enables .NET developers to access classes and their associated properties and methods in the JADE database.

This is achieved by generating a set of classes in .NET that act as proxies for the actual JADE classes. Accessing the proxy classes is similar to the use of the actual classes in JADE.

The .NET library containing the classes is built by exposing the classes (and their methods and properties) of interest in JADE. The wizard used to expose JADE features generates C# class files that you can then build into a re-usable .NET class library.

Exposing JADE Classes

You can use .NET to write the application-related components of your system and access JADE database-related components. The approach consists of the following steps.

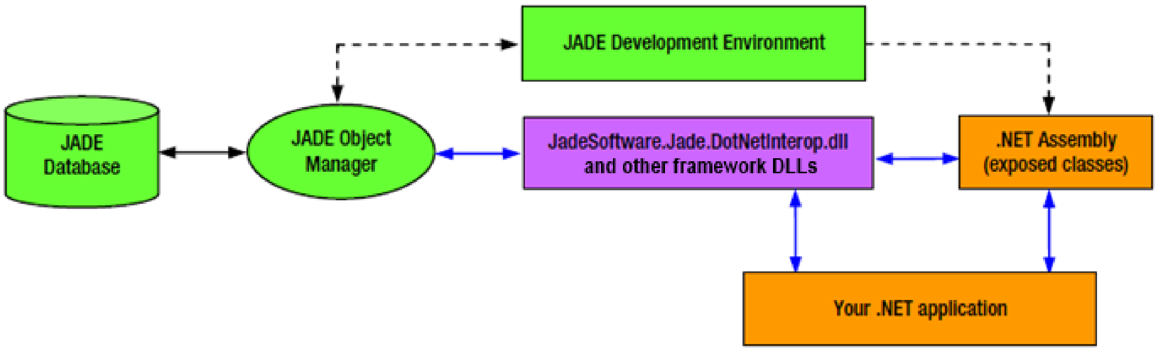
1. Use JADE to define the classes for your .NET application.
2. Define a .NET exposure and generate C# source files as the starting point for building the application. For details, see "Using the C# Exposure Wizard", in Chapter 18 of the *JADE Development Environment User's Guide*.

   The C# class definitions contain the properties and methods defined in the exposure that are required to access them in the JADE database.
3. Use the generated C# Visual Studio project file (or create your own) to build a Dynamic Link Library (assembly) that contains the proxy classes.
4. Add a reference to the assembly, generated in the previous step of this instruction, to your own project.
5. Add instructions to your .NET code relating to JADE connection, JADE class access, and transaction control.
6. Build and run your .NET project.

JADE .NET Framework

To use the JADE .NET API, you must have the tools to build .NET projects; for example, Microsoft Visual Studio 2010 or later. (Microsoft Visual Studio 2017 is recommended.)

The JADE .NET API is implemented in a number of assemblies that provide a set of .NET classes. The following diagram shows the communication and interoperability role played by **JadeSoftware.Jade.DotNetInterop.dll** and other assemblies in the framework.


JADE Development Environment
JADE Database
JADE Object Manager
JadeSoftware.Jade.DotNetInterop.dll and other framework DLLs
.NET Assembly (exposed classes)
Your .NET application

JADE .NET API Documentation

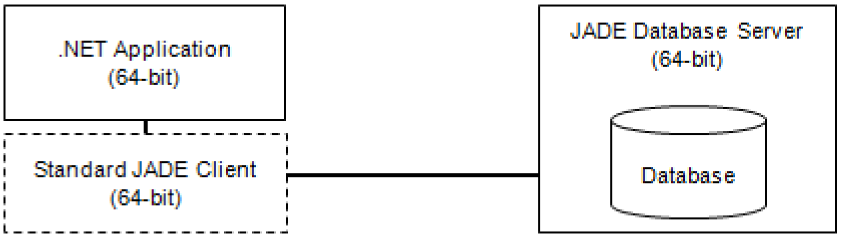
The **JadeDotNetAPI.chm** file, which is located in the installed JADE **documentation** directory (for example, **C:\Jade\JADE Docs\documentation**), contains documentation of the .NET classes and other components, including those for collection concurrency, that comprise the JADE .NET API.

.NET and JADE

The JADE object model includes a class hierarchy that is similar to that of .NET. This common object-oriented approach means there is a natural fit between JADE and .NET, enabling .NET objects to be modeled from the JADE database.

How a .NET Application Connects to JADE

A .NET runtime application connects to a JADE system as a standard JADE client using the **JadeSoftware.Jade.DotNetInterop.dll** library file on Windows. For more details about the code that causes the sign on to occur, see "Connecting to JADE", in Chapter 7.


.NET Application (64-bit)
Standard JADE Client (64-bit)
JADE Database Server (64-bit)
Database

The **JadeSoftware.Jade.DotNetInterop.dll** file is one of the files supplied from JADE as part of a standard JADE client installation, and is typically located in the **bin** directory; for example, **C:\Jade\bin**.

Chapter 6 .NET Exposure

This chapter covers the following topics.

Overview

JADE classes, properties, methods, and class constants are exposed using the .NET Exposure Wizard in the JADE development environment. For details, see Chapter 17 of the *JADE Development Environment User's Guide*.

The exposure generates C# class files that correspond to the exposed JADE classes. These exposed classes are then built into a .NET class library. You can then use this class library in .NET applications to access, create, and update JADE objects in the JADE database.

The generated class library and your application can use the JADE .NET application programming interface (API) contained in the **JadeSoftware.Jade.DotNetInterop.dll** file. This API uses the JADE .NET API to access the JADE database.

Any .NET application or library using the JADE .NET API must include references to the following DLLs that are found in your JADE binary directory (that is, **bin**).

- JadeSoftware.Joob.dll
- JadeSoftware.Joob.Common.dll
- JadeSoftware.Jade.DotNetInterop.dll

.NET Exposures

The **Exposures** command from the Browse menu in the JADE development environment enables you to define a .NET exposure. For details about using the .NET Exposure wizard, see "Using the .NET Exposure Wizard", in Chapter 17 of the *JADE Development Environment User's Guide*.

A .NET *exposure definition* is a collection of JADE classes, methods, and properties that are generated to create a .NET class library for an existing JADE schema.

When a JADE .NET exposure is generated, the following files are created.

File	Contains …
Exposure-name.csproj	A simple C# project file that includes all class files and creates a C# library with the namespace *Exposure-name*.
Exposure-name.config	An example application configuration file that can be used without modification for a test application or to provide code fragments for production configuration files.
Class-name.cs	For each exposed class, a C# class file is created containing the class definition in the namespace *Exposure-name*.
JoobContextExtensions .cs	JoobContext extension methods for creating instances of any exposed class that have a **create** method with parameters, which allows these objects to be created on a specific JoobContext.

You can open the ***Exposure-name*.csproj** C# project file using the professional or express editions of Microsoft Visual Studio 2017 and create the library file ***Exposure-name*.dll**, which defines the namespace *Exposure-name* containing the exposed classes. You can then include this DLL as a reference in a .NET application project, to enable access to the JADE classes.

Application Configuration File

The following is an example of a generated configuration file.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="joob"
             type="JadeSoftware.Joob.Configuration.JoobConfigurationSection,
             JadeSoftware.Joob" />
  </configSections>

  <connectionStrings>
    <add name="myDefault" providerName="JadeSoftware.Joob.JadeConnection"
         connectionString="DataSource=C:/Jade7009/system/;
                           ConfigFile=C:/Jade7009/system/jade.ini;
                           SingleUser=False;
                           Schema=BankingViewSchema;
                           Application=DotNetConnection;
                           IntegratedSecurity=true"></add>
  </connectionStrings>

  <joob defaultConnection="myDefault">
    <installation directory="C:\Jade7009\bin" />
  </joob>

  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/>
  </startup>
</configuration>
```

Within the **<configSections>** element there is a **joob** section, which defines the JADE configuration and is required if you intend using JADE configuration parameters. Within the **joob** section are a number of options that enable you to customize your application.

In the <**connectionStrings**> section, specify the JADE database to which the connection is to be established. The <**add**> sub-element has a **connectionStrings** attribute, which contains a series of comma-separated values that define connection attributes for the JADE database.

- **DataSource**. The location of the folder containing the database control file (**_control.dat**).
- **ConfigFile**. The full path name of the JADE initialization file.
- **SingleUser**. **True** or **False**, depending on the mode in which the database has been opened. The default value is **False**.
- **Schema**. A user schema in the database.
- **Application**. An application defined in the database. The default value is **RootSchemaApp**.
- **IntegratedSecurity**. **True** or **False**, depending on whether the connection uses the current Windows identity established on the operating system thread to log on to JADE.

  The default value is **True**. If you specify **False**, you must add **UserName=** and **Password=** values.

Specifying an application causes the JADE **initialize** or **finalize** methods defined on that application to be invoked. (The default application, **RootSchemaApp**, does not have **initialize** or **finalize** methods defined.)

Specifying an application causes the JADE **initialize** and **finalize** methods defined on that application to be invoked. (The default application, **RootSchemaApp**, does not have **initialize** or **finalize** methods defined.)

Connection String Extensions

You can specify the intended target database in connection strings in one of two ways.

- The existing action; that is, the database control file path and the **ServerNodeSpecifications** parameter value in the [JadeClient] or [ConnectionParams] section of the JADE initialization file.
- Specifying the server URI, which has the following general format.

  ```
  scheme://host-name:port-number or base-name/environment-UUID/
              [server-UUID][?parameters]
  ```

  The *scheme* value specifies the transport type, which can be one of the following values.

  - TcpIp
  - TcpIpv4
  - TcpIpv6
  - HPSM
  - JadeLocal

  The **host-name:port-number or base-name** value specifies the target address of the server.

  The **environment-UUID** value specifies the database, which is expected to be at the target address. This is optionally followed by the database server identity UUID.

  The optional **parameters** values enable you to specify the local address or port for TCP/IP transports.

Tip Connection establishment is fastest when your **connectionString** uses a server URI with both the environment and server UUIDs specified.

The following are examples of **connectionString DataSource** values using:

- TCP/IP version 4 implicitly

  ```
  "DataSource=TcpIp://localhost:6005/48cf13df-bf6d-df11-87e2-2e5925024153;…"
  ```

- TCP/IP version 4, explicitly declaring an explicit local address

  ```
  "DataSource=TcpIpv4://host.company.com:6005/48cf13df-bf6d-df11-87e2-
  2e5925024153?localHostname=aHostName;"
  ```

- TCP/IP version 6, declaring an explicit local port

  ```
  "DataSource=TcpIpv6://[fe81::6fe:7fff:fe97:bd20]:6005/48cf13df-bf6d-df11-87e2-
  2e5925024153?localPort=54321;..."
  ```

- HPSM, declaring both environment and server UUIDs

  ```
  "DataSource=HPSM://localhost:SRCJoob-HPSM/48cf13df-bf6d-df11-87e2-
  2e5925024153/48cf13df-bf6d-df11-87e2-2e5925024153;..."
  ```

Exposed JADE Classes

Each exposed JADE class is:

- Generated as a single ***Class-name*.cs** C# class file
- Generated in the namespace *Exposure-name*
- Inherits from **JoobObject** as the base class
- Contains a constructor to create an object as a transient or persistent JADE object
- Contains a constructor to create an object with the default persistence of the JADE object
- Contains exposed properties, methods, and constants

If the exposed JADE class is a subclass of another exposed JADE class, the class hierarchy is retained in the C# definition; for example, if **SubClass1** is a subclass of **Class1** in the JADE definition, the C# class **SubClass1** is derived from **Class1**.

Note With the addition of **create** methods with parameters in JADE 2018 and higher, the constructors of exposed classes will change, depending on the **create** method of classes. If an exposed JADE class has a **create** method with parameters, the two constructors defined in the previous paragraph also include parameters to match the **create** method of the class.

The data type of these parameters is based on the JADE type. To work correctly, any non-primitive parameters require their class to be exposed. The conversions for JADE primitive types are the same as those for properties. For details, see "Exposed JADE Properties", in the following section.

The following example shows an exposed JADE class.

```
[System.Runtime.Serialization.KnownTypeAttribute(typeof(Tender))]
[System.Runtime.Serialization.DataContractAttribute(IsReference=true)]
[System.ComponentModel.DataAnnotations.MetadataTypeAttribute(typeof
     (TenderSaleMetadata))]
[JadeSoftware.Joob.Client.JoobClassAttribute("TenderSale",
     "RootSchema.CommonSchema.ErewhonInvestmentsModelSchema",
     ClassNamespace="ErewhonTesting")]
```

```
[JadeSoftware.Joob.Metadata.JomlTypeAttribute
    (JadeSoftware.Joob.Metadata.JomlTypeKind.Class, "TenderSale", typeof
    JoobObject))]
public partial class TenderSale: Sale
{
    private static TenderSaleMetadata _metaModel;
    partial void _initialize();
    static TenderSale()
    {
        _metaModel = MetadataCache<TenderSaleMetadata>.GetData(null);
    }
    public TenderSale():
        this(JadeSoftware.Joob.ClassPersistence.Persistent)
    {
    }
    public TenderSale(JadeSoftware.Joob.ClassPersistence lifetime):
        base(lifetime, typeof(TenderSale), _metaModel.metaClass)
    {
        this._initialize();
    }
    protected TenderSale(JadeSoftware.Joob.ClassPersistence lifetime,
        System.Type type, JadeSoftware.Joob.ClassMetadata metaClass):
        base(lifetime, type, metaClass)
    {
        this._initialize();
    }

#region Jade Properties
#region Jade Methods
#region Jade Constants
}
```

The generated classes are *decorated* with a number of attributes, which are used by the JADE engine to ensure correct and consistent access to the class in the JADE database. For each exposed class, an additional class is generated, which is used internally to ensure consistency between the JADE database definition and the C# definition. The name of this class is formed by appending **Metadata** to the name of the exposed class. For example, if the **TenderSale** class is exposed, the class is named **TenderSaleMetadata**.

Exposed JADE Properties

Each exposed JADE property is defined in the C# class in the **Jade Properties** region. Each property has an accessor and mutator (that is, a **get** method and a **set** method).

The data type of the property is based on the JADE type. The conversions for JADE primitive types are listed in the following table.

JADE Type	.NET Type
Binary	Byte[]
Boolean	Boolean
Byte	Byte
Character	Char

JADE Type	.NET Type
Date	DateTime
Decimal	Decimal
Integer	Int32
Integer64	Int64
JadeBytes	JoobBytes
MemoryAddress	<not supported>
Point	Point (defined in **JadeSoftware.Jade.DotNetInterop**)
Real	Double
String	String
StringUtf8	String
Time	TimeSpan
TimeStamp	DateTime
TimeStampInterval	TimeSpan
TimeStampOffset	DateTimeOffset

The generated code varies depending on the JADE property type. The following example shows an exposed JADE property.

```
[JadeSoftware.Joob.Client.JoobPropertyAttribute("myTender", typeof(Tender),
DatabaseTypeName="Tender")]
[System.Runtime.Serialization.DataMemberAttribute()]
public Tender MyTender
{
    get
    {
        return this.GetPropertyReference<Tender>(_metaModel.myTender);
    }
    set
    {
        this.SetPropertyReference(_metaModel.myTender, value, false);
    }
}
```

Exposed JADE Methods

Each exposed JADE method is defined in the C# class in the **Jade Methods** region.

The data type of the return value and parameters is based on the JADE type. The conversions for JADE primitive types are the same as those for properties. For details, see "Exposed JADE Properties", in the previous section.

JADE Method Parameter Usage

In the signature of a JADE method, the type of each parameter must be declared along with the **Usage** value, which can be **Usage.Constant**, **Usage.Input**, **Usage.IO** (combination of input and output), or **Usage.Output**.

If you do not specify a usage, a default of **Usage.Constant** is assumed.

The parameter usages are as follows.

- For **Usage.Constant** and **Usage.Input** parameters, the parameter value specified in the method call is passed to the corresponding parameter in the called method.

  Note You cannot assign to a **Usage.Constant** or **Usage.Input** parameter.

- For **Usage.Output** parameters, the value is passed in the reverse direction, from the parameter of the called method back to the corresponding parameter of the caller. This copying back of the parameter value occurs when the called method returns.
- For **Usage.IO** parameters, the parameter value is passed in both directions, as follows.
  - From the caller to the called method when the method begins
  - From the called method back to the caller when it returns

The following is an example of an exposed JADE method.

```
[JadeSoftware.Joob.Client.JoobMethodAttribute("addIntegers")]
public Int32 AddIntegers(Int32 i1, Int32 i2)
{
    using (JadeParam retnParam = new JadeParamInteger(Usage.Output),
                     jadeParam1 = new JadeParamInteger(i1),
                     jadeParam2 = new JadeParamInteger(i2))
    {
        this.SendMessage(_metaModel.addIntegers,
                         retnParam,
                         jadeParam1,
                         jadeParam2);
        return (retnParam as JadeParamInteger).Value;
    }
}
```

Exposed JADE Class Constants

Each exposed JADE class constant is defined in the C# class in the **Jade Constants** region.

The data type of the constant is based on the JADE type. The conversions for JADE primitive types are the same as those for properties. For details, see "Exposed JADE Properties", earlier in this chapter.

Collections

Collection classes are derived from a subclass of various classes in the **JadeSoftware.Joob** namespace listed in the following table, to inherit the required behavior.

Collection	Description
JadeSoftware.Joob.ObjectArray	Array of JADE objects
JadeSoftware.Joob.ObjectSet	Set of JADE objects
JadeSoftware.Joob.MemberKeyDictionary	Dictionary of JADE objects with one or more keys that are in the member class

Collection	Description
JadeSoftware.Joob.ExtKeyDictionary	Dictionary of JADE objects with one or more keys that are externally supplied
JadeSoftware.Joob.DynamicDictionary	Dictionary of objects that encapsulates the behavior of the JADE **DynaDictionary** class

The following example shows an exposed JADE **Collection** class called **AgentDictByName** that contains JADE objects of type **Agent**.

```
public partial class AgentDictByName:
                     MemberKeyDictionary<AgentDictByNameKey, Agent>
{
    private static AgentDictByNameMetadata _metaModel;
    partial void Initialize();
    static AgentDictByName()
    {
        _metaModel = MetadataCache<AgentDictByNameMetadata>.GetData(null);
    }
    private AgentDictByName():
            this(JadeSoftware.Joob.ClassPersistence.Transient)
    {
    }
    public AgentDictByName(JadeSoftware.Joob.ClassPersistence lifetime):
           base(lifetime, typeof(AgentDictByName), _metaModel.metaClass)
    {
        this.Initialize();
    }
    protected AgentDictByName(JadeSoftware.Joob.ClassPersistence lifetime,
              System.Type type, JadeSoftware.Joob.ClassMetadata metaClass):
              base(lifetime, type, metaClass)
    {
        this.Initialize();
    }
    public virtual Agent this[String name, DateTime dob]
    {
        get
        {
            AgentDictByNameKey key = new AgentDictByNameKey(name, dob);
            return base[key];
        }
        set
        {
            AgentDictByNameKey key = new AgentDictByNameKey(name, dob);
            base[key] = value;
        }
    }
    public virtual bool TryGetValue(String name, DateTime dob,
                                    out Agent value)
    {
        AgentDictByNameKey key = new AgentDictByNameKey(name, dob);
        return base.TryGetValue(key, out value);
    }
    public virtual bool TryGetValue(String name, DateTime dob,
           JadeSoftware.Joob.SearchStrategy strategy, out Agent value)
    {
        AgentDictByNameKey key = new AgentDictByNameKey(name, dob);
        return base.TryGetValue(key, strategy, out value);
    }
    public virtual IJoobDictionaryEnumerable<AgentDictByNameKey, Agent>
           StartingAtKey(String name, DateTime dob)
    {
        AgentDictByNameKey key = new AgentDictByNameKey(name, dob);
        return base.StartingAtKey(key);
    }
    public virtual IJoobDictionaryEnumerable<AgentDictByNameKey, Agent>
           StartingAtKey(String name, DateTime dob,
           JadeSoftware.Joob.SearchStrategy strategy)
    {
        AgentDictByNameKey key = new AgentDictByNameKey(name, dob);
        return base.StartingAtKey(key, strategy);
    }
    public virtual void Remove(String name, DateTime dob)
    {
        AgentDictByNameKey key = new AgentDictByNameKey(name, dob);
        base.Remove(key);
    }
    public virtual void Remove(String name, DateTime dob, Agent member)
    {
        AgentDictByNameKey key = new AgentDictByNameKey(name, dob);
        base.Remove(key, member);
    }
    public virtual bool ContainsKey(String name, DateTime dob)
    {
        AgentDictByNameKey key = new AgentDictByNameKey(name, dob);
        return base.ContainsKey(key);
    }
}
```

In the following example, **root** contains a reference **AllAgents** of type **AgentDictByName**.

```
foreach (Agent agent in root.AllAgents)
{
    agent.DoSomething();
}
```

When indexing a JADE array in .NET, the .NET convention of zero-based arrays is used; that is, the element at position zero (**0**) is the first element in the array. In the following example, a C# class called **SomeClass** has an **IntegerArray** property called **MyIntegerArray**.

```
SomeClass obj = joobContext.FirstInstance<SomeClass>();
Int32 i = obj.MyIntegerArray[0];     // returns first element of array
Int32 j = obj.MyIntegerArray.At(0);  // also returns first element of array
```

In the following example, **root** contains a reference **AllAgents** of type **AgentDictByName**.

```
foreach (Agent agent in root.AllAgents)
{
    agent.DoSomething();
}
```

When indexing a JADE array in .NET, the .NET convention of zero-based arrays is used; that is, the element at position zero (**0**) is the first element in the array. In the following example, a C# class called **SomeClass** has an **IntegerArray** property called **MyIntegerArray**.

```
SomeClass obj = joobContext.FirstInstance<SomeClass>();
Int32 i = obj.MyIntegerArray[0];     // returns first element of array
Int32 j = obj.MyIntegerArray.At(0);  // also returns first element of array
```

Chapter 7 Developing Applications in .NET to Use JADE Classes

This chapter covers the following topics.

Overview

This chapter documents classes in the JADE .NET API that you can use to build .NET applications that work with the JADE classes in an exposure DLL. For more details, see the JADE .NET API documentation (that is, the **JadeDotNetAPI.chm** file in the installed JADE **documentation** directory; for example, **C:\Jade\JADE Docs\documentation**).

Using JADE Classes in .NET

The **JoobContext** and **JoobConnection** classes in the **JadeSoftware.Joob** library provide the link between a .NET application and a JADE system.

Any .NET assembly using the JADE .NET API must include references to the following DLLs that are found in the JADE binary (**bin**) directory.

- JadeSoftware.Jade.DotNetInterop.dll
- JadeSoftware.Joob.dll
- JadeSoftware.Joob.Common.dll

In addition, a reference to the exposure DLL created using the .NET Exposure wizard in the JADE development environment is required. For details about the exposure DLL, see ".NET Exposure", in Chapter 6.

To simplify access to these classes in a C# project, add the following **using** directives.

```
using JadeSoftware.Jade.DotNetInterop;
using JadeSoftware.Joob.Client;
using JadeSoftware.Joob;
using <namespace-for-exposed-classes>;
```

JADE .NET Assembly Public Keys

JADE .NET assemblies are signed. Each release uses the same public key token for its assemblies, but they have different versions, as shown in the following table.

Build/Configuration	PublicKeyToken
Release_Ansi|x64	4ffc2b9eb5630e47
Release_Unicode|x64	b5033a0291fb93d9

An example of the **app.config** file is as follows.

```
<configSections>
    <section name="joob"
        type="JadeSoftware.Joob.Configuration.JoobConfigurationSection,
        JadeSoftware.Joob, Version=7.0.9.0, Culture=neutral,
        PublicKeyToken=4ffc2b9eb5630e47">
    </section>
</configSections>
```

An application run with this example **app.config** file requires **JadeSoftware.Joob.dll** with an assembly version of 7.0.9.0 and signed with the **Release_Ansi|X64** key.

Hot fix versions of **JadeSoftware.Joob.dll** (and so on) are released with the same assembly version as the initial consolidated release (for example, 7.0.9.0) but the file version indicates the hot fix number. Your assemblies, therefore, do not need to be rebuilt when a hot fix is installed but they must be rebuilt when a subsequent consolidated release is installed.

Connecting to JADE

A **JoobContext** object is used to connect to the database. The default no-parameters constructor uses the **defaultConnection** information specified in the application configuration file. For details, see "Application Configuration File", in Chapter 6.

The following syntax creates a context object using the no-parameters constructor and establishes a connection to the JADE database.

```
JoobContext context = new JoobContext();
```

The following syntax establishes a connection using an alternative connection string defined in the application configuration file.

```
JoobContext context =
    new JoobContext(JoobConnectionStringBuilder.CreateFromConfig("other"));
```

When a connection is no longer required, the **JoobContext** object should be disposed of to release the connection back to the pool.

```
context.Dispose();
```

If a connection is required only at certain times in the life of the application, you can create a temporary connection with a **using** statement. The **JoobContext** object is implicitly disposed of at the end of the **using** block.

```
using (JoobContext context = new JoobContext())
{
    // access the database
}
```

For details about using non-default connections and specifying connection parameters from your logic, see the JADE .NET API documentation (that is, the **JadeDotNetAPI.chm** file in the installed JADE **documentation** directory; for example, **C:\Jade\JADE Docs\documentation**).

Pool of Available Processes

When a connection is first established from a .NET application to JADE, a node is created with a pool of available processes. The connection uses one of the processes and releases it back to the pool when the connection is closed, which happens when the associated **JoobContext** object is disposed of.

If the default connection is used, the number of processes in the pool can be specified in the **defaultPoolSize** attribute of the **<jade>** element of the application configuration file, as shown in the following example.

```
<joob defaultConnection="myDefault" defaultPoolSize="15">
```

If the **defaultPoolSize** attribute is omitted, the default value of **10** is used.

Note If you use the free JADE developer licence, which allows five processes only, reduce the pool size as follows.

```
<joob defaultConnection="myDefault" defaultPoolSize="2">
```

If you do not change the default value, you will encounter exception 5504 (*You have exceeded the number of Process Licenses)* when creating a **JoobContext** object.

For an alternative connection string defined in the application configuration, you can override the **defaultPoolSize** value with the **poolSize** attribute within the **<tuning>** element, as shown in the following example.

```
<tuning>
    <add connection="other" poolSize="20" />
</tuning>
```

Accessing Database Instances

The **JoobContext** class provides methods to access database objects from a specified class.

Method	Returns …
FirstInstance<T>	The first instance of a class
FindInstance<T>	An object with the specified OID
LastInstance<T>	The last instance of a class
AllInstances<T>	All instances of a class

The following typical sequence of C# calls returns the first instance of the **Company** class, iterates through a collection on the class, and uses references and methods of the objects in the collection.

```
JoobContext context = new JoobContext();
Company company = context.FirstInstance<Company>();
```

If you know the object identifier (OID) of a JADE database object, you can obtain a reference to that object in your C# code, as shown in the following example.

```
JoobContext context = new JoobContext();
ObjectId oid = new ObjectId(3272, 1);
Agent comp = context.FindInstance<Agent>(oid);
```

In the following example, a virtual collection of instances of the **Stock** class is iterated.

```
JoobContext context = new JoobContext();
foreach (Agent agent in context.AllInstances<>(Agent))
{
    // process agent
}
```

Accessing JADE System Objects

You can obtain the OIDs of the JADE system objects by using the **GetSystemVariables** method of the **JoobContext** instance, which returns an instance of the **SystemVariables** class. The properties of a **SystemVariables** object are the OIDs of the JADE system objects. They are listed in the following table.

JADE System Object	Type	Description
App	Application subclass (which must be exposed)	The current transient application instance. This is often used to store application-specific data.
Currentschema	JadeSoftware.Joob.MetaSchema.Schema	Current user-defined schema.
Global	Global subclass (which must be exposed)	The persistent object for a schema, which is shared by all applications running from that schema. This is often used to exchange information between applications or to retain information when an application closes.
Node	JadeSoftware.Joob.Management.JoobNode	The workstation that hosts the execution of the current process. One node object exists for each *logical* workstation connected to the server node workstation. There is one fixed server node and none or many client nodes. A node represents a workstation that runs a number of processes.

JADE System Object	Type	Description
Process	JadeSoftware.Joob.Management.JoobSession	The current thread in a workstation executing the current method.
RootSchema	JadeSoftware.Joob.MetaSchema.Schema	The JADE **RootSchema**.
System	JadeSoftware.Joob.Management.JoobSystem	The JADE architectural environment consisting of a group of nodes to which the current node belongs.

Notes You must expose the classes of the **App** and **Global** system objects for the types to be available to your C# code.

See also "JoobContexts, Sessions, and JoobConnections", in Chapter 2, for details about the C# wrappers for the JADE Object Manager **Process**, **Node**, and **System** classes.

The following code would obtain a reference to the **Node** object.

```
JoobContext context = JoobContext.CurrentContext;
SystemVariables sv = context.GetSystemVariables();
ObjectId nodeOid = sv.Node;
JadeSoftware.Joob.Management.JoobNode
    node = context.FindInstance<JadeSoftware.Joob.Management.JoobNode>(nodeOid);
```

A typical use of the **App** object is to store a reference to a *root* object, which is a singleton persistent object that contains collections of all objects in a class. For example, in a banking application, the root object would represent the bank itself and would have a collection of all of the customers of the bank. The following code would obtain a reference to the **App** object.

```
JoobContext context = JoobContext.CurrentContext;
SystemVariables sv = context.GetSystemVariables();
ObjectId appOid = sv.App;
ErewhonInvestmentsModelApp
    app = context.FindInstance<ErewhonInvestmentsModelApp>(appOid);
```

Transient **App** objects are created when a connection to JADE is first established and the pool of processes is created (typically when an application first creates a **JoobContext** instance). An **App** object is associated with each process in the pool. Each **App** object exists for the lifetime of the process and is retained even when a process is released back to the pool (which happens when the associated **JoobContext** object is disposed of).

Because the **App** object for each process is retained, it is available when a **JoobContext** instance is next created and associated with that process. However, the associated process may be any of the processes in the pool.

If an **initialize** method is defined for the database application specified in the connection string, it is invoked when each **App** object is created (typically when the first connection to JADE is established). Similarly, if a **finalize** method is defined, it is invoked when each **App** object is deleted (typically when the .NET application terminates).

Creating Objects

You can use the **new** operator to create a persistent or transient instance of an exposed class. The constructor to use is determined by whether the exposed JADE class has a **create** method with parameters.

For an exposed class where the **create** method has no parameters, the **new** operator is used together with the no-parameters constructor for the class or the constructor with a **ClassPersistence** enumeration value of **Persistent** or **Transient**; for example:

```
Agent agent = new Agent();
Agent agent = new Agent(ClassPersistence.Persistent);
```

Alternatively, you can use the **CreateInstance** method of the **JoobContext** class.

```
JoobContext context = JoobContext.CurrentContext;
Agent agent = context.CreateInstance<Agent>(ClassPersistence.Persistent);
```

For an exposed class where the **create** method has parameters, the **new** operation is used together with the constructor with the matching parameters as the **create** method or the constructor with matching parameters and a **ClassPersistence** enumeration value of **Persistent** or **Transient**; for example:

```
Customer customer = new Customer("Wilbur", "wilbur@jadeworld.com");
Customer customer = new Customer(ClassPersistence.Persistent, "Wilbur",
                    "wilbur@jadeworld.com");
```

Note Attempting to use the no-parameters constructor on a class requiring parameters raises exception 4027 (*Method called with incorrect number of parameters*).

Alternatively, you can create an instance of an exposed class by using the **JoobContext** extension methods generated in **JoobContextExtensions.cs**. Each exposed class that has a **create** method with parameters generates a **JoobContext** extension method with the signature **Create*XX*Instance**, with the ***XX*** value being the name of the exposed class; for example:

```
JoobContext context = JoobContext.CurrentContext;
Customer customer = context.CreateCustomerInstance(ClassPersistence.Persistent,
                    "Wilbur", "wilbur@jadeworld.com");
```

Transient objects belong to the process that created them and are not accessible by any other process. The transient objects for a process are retained until explicitly deleted or when the process is disposed of (typically when the .NET application terminates).

Because the transient objects for each process are retained, they are available the next time a **JoobContext** instance is created and associated with that process. However, the associated process may be any of the processes in the pool.

Note To create, modify, or delete a persistent object, you must be in transaction state. For details, see "Transactions", later in this chapter.

Deleting Objects

You can use the **Delete** method, which is inherited from the **JoobObject** class, to delete an instance of an exposed JADE class.

```
agent.Delete();
```

Alternatively, you can use the **DeleteObject** method of the **JoobContext** class.

```
JoobContext context = JoobContext.CurrentContext;
context.DeleteObject(agent);
```

Note To create, modify, or delete a persistent object, you must be in transaction state. For details, see "Transactions", later in this chapter.

Locking

The locking-related methods provided by the **JoobContext** class are shown in the following table.

Method	Description
Lock	Locks a JADE object and throws an exception if the attempt fails
Unlock	Releases a lock on a JADE object
TryLock	Locks a JADE object and returns **False** if the attempt fails
GetLockStatus	Gets the type and duration of the lock on a JADE object by the current process

In the following example, an attempt is made to lock an **agent** object using the **Lock** method. If the object is already locked with an incompatible lock, a normal lock exception is thrown. The JADE concepts of *passback*, *continue*, *resume*, or *abort* (for the action that is taken after the exception handling code has been executed) are not supported.

```
JoobContext context = JoobContext.CurrentContext;
Agent agent = context.FirstInstance<Agent>();
try
{
    context.Lock(agent,                      // object to be locked
                 LockType.Exclusive,         // type of lock
                 LockDuration.Session,       // session/transaction duration
                 TimeSpan.FromSeconds(5));   // time before exception thrown
}
catch (JoobObjectLockedException)
{
                                             // lock object handling code
}
```

Notifications

The **JoobContext** class provides the **RegisterNotificationHandler** method to request being notified about events for a specified JADE object and the **RegisterClassNotificationHandler** method to request being notified about events for all objects of a specified class.

The first parameter is the *target* of the notifications. The second parameter is the event type number (you can use the constants in the **NotificationEventConstants** class for system events). The third parameter is the name of the event handler method to be invoked.

The following example shows how you can register for notifications.

```
JoobContext context = JoobContext.CurrentContext;
// Register for any kind of system event on the 'agent' object
// the event handler method is called 'myEventHandler'
context.RegisterNotificationHandler(agent,
                                    NotificationEventConstants.SystemAny,
                                    myEventHandler);

// Register for user event 5000 for all instances of the Agent class
// the event handler method is called 'event5000Handler'
context.RegisterClassNotificationHandler(agent, 5000, event5000Handler);
```

There are **UnregisterNotificationHandler** and **UnregisterClassNotificationHandler** methods to stop the sending of notifications. The methods have the same parameters as the methods that register for notifications.

The **Notify** method, which is inherited from the **JoobObject** class, publishes a user event. The first parameter is the event type number. The second parameter is **True** if the event is raised immediately and **False** if it deferred until the next time a transaction is committed. The method is overloaded so that the third parameter, which is user information about the event, can be of any type.

The following example shows how you can cause a user event.

```
// 'agent' is an instance of the Agent class
agent.Notify(agent, false, 0);
```

An event handler method has a signature similar to any C# event handler.

```
void myEventHandler(object sender, NotificationEventArgs e)
{
    JoobContext context = sender as JoobContext;
    int eventNo = e.EventNo;
    ObjectId target = e.Target;
}
```

The first parameter is **JoobContext** object that was used to register for notifications. The second parameter is a **NotificationEventArgs** object, which has the following properties.

Property	Description
EventNumber	The event number integer that was passed as the middle parameter of the **RegisterClassNotificationHandler** or **RegisterNotificationHandler** method.
Target	The **ObjectId** of the object to which the event happened.
UserInfo	For user events, it is an object representing the **UserInfo** parameter that was passed to the **Notify** method when the user event was published. For system events, it is a null object reference.

Notifications in JADE and .NET

If a .NET application calls a JADE method that executes the **beginNotification** or **beginClassNotification** method, the notification is delivered to the thread that created the **JoobContext** object.

When the **beginNotification** or **beginClassNotification** is executed in a JADE method, it is regarded as subscribing to a different notification than one done by executing the **RegisterNotificationHandler** or **RegisterClassNotificationHandler** method in .NET, even if all of the parameters are equivalent. However, when an event is explicitly (or implicitly) caused in either language, it is delivered to all subscribers even if the language that caused the notification did not subscribe to the notification.

When a **JoobContext** object is disposed of, all subscriptions are cancelled.

As the JADE code is running within .NET, the JADE process will not have an idle state. Consequently, you must execute the **doWindowEvents** method of the **Application** class to enable the notification response method to run. For details about the cautions regarding the use of this method, see Volume 1 of the *JADE Encyclopaedia of Classes*.

Exceptions

The **JadeSoftware.Joob.Exception** namespace contains exception classes, including JOM exceptions that are implemented as subclasses of **JoobJomException**. These exceptions can be caught in the normal manner, as shown in the following example.

```
void catchAnException()
{
    JoobContext context = JoobContext.CurrentContext;
    Client c = context.FirstInstance<Client>();

    try
    {
        c.Name = c.Name; // generated 'update outside transaction' exception
    }
    catch (JoobUpdateOutsideTransactionException)
    {
       // Expected exception
    }
    catch (JoobJomException jje)
    {
        int jomErrorError = jje.ErrorCode;
        String jomErrorText = jje.ErrorText;
    }
    catch (JoobException e)
    {
        String errorMessage = e.Message;
    }
}
```

Accessing Imported .NET Exception Classes

The **JadeDotNetInvokeException** user interface exception class provides the **dotNetExceptionObject** property, that is a type of **JadeDotNetType**. This property is populated with the .NET **Exception** object when an exception of type **JadeDotNetInvokeException** is generated and the class of the .NET exception object class was imported from the .NET assembly.

If the exception type object is not available, the property value is null. If the class was not imported from the .NET assembly, an object of **JadeDotNetType** type is created.

If the property is set to a reference, you can cast the object to its type and use it to obtain further .NET exception information.

Transactions

The **JoobContext** class provides the **BeginTransaction** method, which returns a transaction object. All JADE actions required to be in transaction state (that is, create, update, or delete actions) must be within the scope of the transaction object returned by the **BeginTransaction** method. The transaction object provides the methods **Commit** and **Rollback**, to commit or abort the transaction, respectively.

The following example shows a method that uses a transaction to delete a persistent **Agent** object.

```
void DeleteClient(Client client)
{
    JoobContext context = JoobContext.CurrentContext;
    using (System.Data.IDbTransaction tx = context.BeginTransaction())
```

```
        {
            client.Delete();
            tx.Commit();
        }
    }
```

If the statement that commits the transaction is omitted, the transaction is rolled back when the transaction object is disposed of, at the end of the using block.

Multiple Database Access for .NET

The JADE .NET API enables access to multiple JADE databases; that is, a single .NET application (operating system process) can access a number of JADE databases concurrently.

The restrictions on accessing multiple JADE databases are:

- The JADE databases must be at the same feature level.
- Multiple connections to the same database are not allowed.
- There is a limit of 20 concurrent connections.
- Some JADE initialization file parameters (for example, the **LogDirectory** parameter in the [JadeLog] section) are read-only when the first database connection is opened, and they remain unchanged for the life of the process.

Tip Use a common JADE initialization file for all database connections, to ensure that the same **jommsg.log** file is used, regardless of the order in which database connections are established.

Using Multiple Database Access

To access multiple JADE databases, you must supply connection details via the appropriate **JoobContext** constructor. This can take the form of an existing **JoobConnection** object, or it can be via a **JoobConnectionStringBuilder** object, as shown in the following example.

```
<BLOCK 1A>
using (JoobContext context = new JoobContext
                      (JoobConnectionStringBuilder.CreateFromConfig("LocalJade")))
{
    // Established context to database with schema MultipleDb1 as
    // specified by the "LocalJade" connection string in the app.config
     <BLOCK 2A>

    using (JoobContext remoteContext = new JoobContext
                      (JoobConnectionStringBuilder.CreateFromConfig("RemoteJade")))
    {
        // Established context to database with schema MultipleDb2 as
        // specified by "RemoteJade" connection string.
         <BLOCK 3>
    }
    <BLOCK 2B>
}
<BLOCK 1B>
```

The **app.config** file reads as follows.

```
<connectionStrings>
  <add name="RemoteJade" providerName="JadeSoftware.Joob.JoobConnection"
    connectionString="DataSource=tcpip://remoteHost:6005/7bf0f9cd-680a-
    e111-9eaf-5ae520524153;ConfigFile=\\remoteHost\C:\Jade\system\jade.ini;
    Schema=MultipleDb2;IntegratedSecurity=True"></add>
  <add name="LocalJade" providerName="JadeSoftware.Joob.JoobConnection"
    connectionString="DataSource=C:\Jade\system;ConfigFile=C:\Jade\system\jade.ini;
    Schema=MultipleDb1;IntegratedSecurity=True"></add>
</connectionStrings>
```

The model for multiple database access follows the pattern of requiring a live **JoobContext** before a **JoobObject** can interact with its database.

When an appropriate **JoobContext** has been instantiated for a database, **JoobObject** instances from that database can be retrieved, created, and de-referenced. Each **JoobObject** instance knows the database to which it belongs and provided that a **JoobContext** is alive for its database on the current thread, **JoobObject** manipulation can occur.

For example, a **JoobObject** variable of a **MultipleDb2** class can be declared in *BLOCK 1A* or *BLOCK 2A*. You can set, get, or invoke JADE schema-defined properties and methods only from within *BLOCK 3*. You can locate instances of classes only from schema **MultipleDb2** within *BLOCK 3*. However, you can manipulate instances of classes from **MultipleDb1** within *BLOCK 2A*, *BLOCK 2B*, and *BLOCK 3*.

Appendix A Mapping JADE Primitives to CLR Data Types

This appendix describes the mapping of JADE primitive types to Common Language Runtime (CLR) data types.

CLR Type	JADE Primitive Type	Conversion Notes
Boolean	Boolean	
Byte	Byte	
Int32	Integer	
Int64	Integer64	
Double	Real	
Decimal	Decimal	CLR decimals are in the range approximately -7.9x10$^{\wedge 28}$ through 7.9x10$^{\wedge 28}$. JADE decimals are in the range -10$^{\wedge 23}$ - 1 through 10$^{\wedge 23}$ - 1. A .NET **Decimal** value has 96 bits and can have up to 29 significant digits. JADE supports **Decimal** properties with values with up to 23 significant digits. If necessary, JADE rounds decimal values on assignment to the number of decimal places defined on the property.
Char	Character	
String	String, StringUtf8	
Char[]	String, StringUtf8	
DateTime	Date, TimeStamp	The time part of a CLR **DateTime** is lost when saving to the database.
DateTimeOffset	TimeStampOffset	
TimeSpan	Time, TimeStampInterval	The JADE **Time** type supports only CLR **TimeSpan** values in the range 0:00:00 through 23:59:59. (There may be an overflow or data loss saving data to the database.)
Byte[]	Binary	

Appendix B JADE .NET Spatial Feature

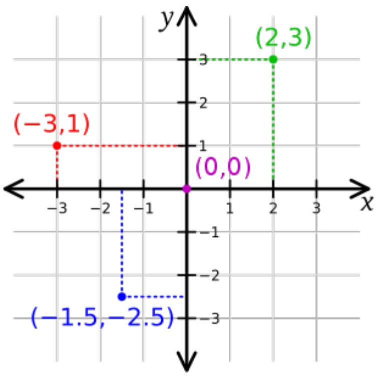
This appendix discusses the spatial feature in JADE .NET, which enables you to develop applications to store, edit, and query spatial information directly through the JADE .NET API.

The first part of this appendix focuses on spatial data, explaining how you can create, edit, and analyze spatial objects in JADE .NET. The second part talks about the spatial index used in JADE .NET, which can be helpful when dealing with a large collection of spatial objects.

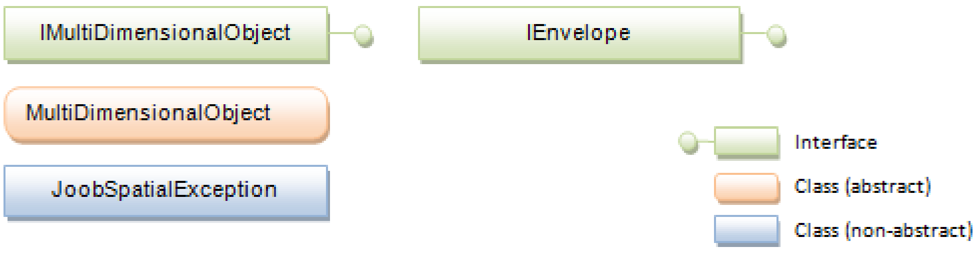
Spatial Data

Spatial data, also known as geospatial data or geographic information, is the data that describes the geographic location of features and boundaries on the earth.

The geometry data type uses the two-dimensional Cartesian coordinate system, meaning that spatial data of this type are presented in a uniform two-dimensional plane with units of your choice. When dealing with this type of spatial data, or planar data, we assume that the earth is flat.



In JADE .NET, the following interfaces and classes are introduced to assist in the development of spatial components within applications. However, they are generalized to a level higher, so that they can also be used in non-spatial domains in the future.



IMultiDimensionObject is an interface representing all objects that have dimensionality – not only spatial geometries, but also other non-spatial dimensional data as well. It exposes two properties (**Dimension** and **Envelope**) and some predicate methods such as **Contains** and **Touches** for evaluating relationships between two instances of **IMultiDimensionalObjects**.

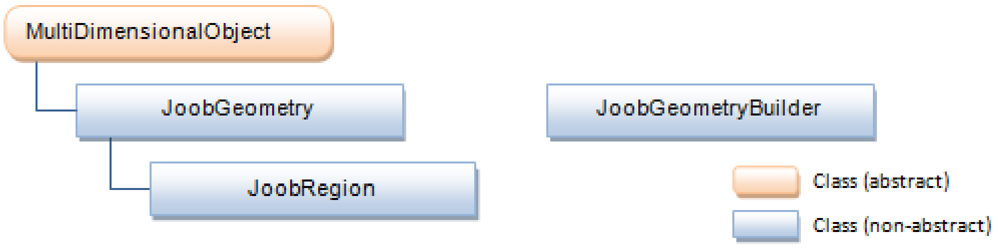
IEnvelope is an interface representing the envelope of an **IMultiDimensionalObject**. **IEnvelope** implements **IMultiDimensionalObject**.

MultiDimensionalObject is an abstract class directly implementing **IMultiDimensionalObject** and it is the base class for **JoobGeometry**, which is discussed in the following section.

Unlike the interfaces and classes in previous paragraphs, **JoobSpatialException** is specific to the spatial domain and is used for reporting errors that occurred during the creation and manipulation of spatial data.

Working with Geometry

As shown in the following diagram, there are a few more classes introduced that enable you to create and manipulate geometries in JADE .NET.



JoobGeometry inherits from **MultiDimensionalObject** and can be used to represent all geometry types that JADE .NET supports; that is, **Point**, **LineString**, **LinearRing**, **Polygon**, **MultiPoint**, **MultiLineString**, **MultiPolygon**, and **GeometryCollection**.

JoobGeometry exposes a large number of properties and methods that are useful for querying geometry information and performing spatial analysis. Their implementations are compatible with the Open Geospatial Consortium's (OGC)'s Simple Features specification version 1.1.0:

- OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 1: Common architecture
- OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 2: SQL option

JoobRegion is a special subclass of **JoobGeometry**, representing the Minimum Bounding Rectangle (MBR) of a specified **JoobGeometry**.

JoobGeometryBuilder, which is described later in this appendix, is a helper class that enables you to create geometries in a point-by-point fashion.

Creating a Geometry

In JADE .NET, there are several ways to create a **JoobGeometry** object, by using the:

- **JoobGeometry** constructor to accept a string encoded in the WKT (Well Known Text) format
- **JoobGeometry** constructor to accept a byte array encoded in the WKB (Well Known Binary) format
- **JoobGeometryBuilder** class

Creating a geometry using WKT or WKB is quite straightforward — all you need to do is to supply a valid WKT string or a WKB byte array to a **JoobGeometry** constructor; for example:

```
public void CreateRing()
{
    var wkt = "LineString (10 20, 30 15, 20 15, 10 10, 10 20)";
    using (var geom = new JoobGeometry(wkt))
    {
        ...
    }
}
```

If an invalid WKT or WKB is provided, a **JoobSpatialException** is thrown. For details about the WKT and WKB formats, see the OGC specification Simple feature access - Part 1: Common. Geometry data do not always present themselves in WKT or WKB formats; instead, they may come with a series of coordinates. In this circumstance, you can use **JoobGeometryBuilder** to create a geometry one point at a time. The following are two examples of using **JoobGeometryBuilder**: the first creates a simple line string starting from the coordinate (10, 40) to the coordinate (30, 50), and the second creates a polygon with one interior ring.

```
public JoobGeometry CreateLineString()
{
    var builder = new JoobGeometryBuilder();
builder.BeginGeometry(GeometryType.LineString);

    builder.BeginFigure(10, 40);
    builder.AddLine(30, 50);
    builder.EndFigure();

    builder.EndGeometry();
    return builder.GetConstructedGeometry(ClassPersistence.Persistent);
}
public JoobGeometry CreatePolygonWithOneInteriorRing()
{
    var builder = new JoobGeometryBuilder();
    builder.BeginGeometry(GeometryType.Polygon);

    builder.BeginFigure(1, 2);
    builder.AddLine(3, 4);
    builder.AddLine(4, 1);
    builder.AddLine(1, 2);
    builder.EndFigure();

    builder.BeginFigure(2, 2);
    builder.AddLine(3, 3);
    builder.AddLine(3, 2);
    builder.AddLine(2, 2);
    builder.EndFigure();

    builder.EndGeometry();
    return builder.GetConstructedGeometry(ClassPersistence.Persistent);
}
```

It is important to note that there is a finite-state machine embedded in **JoobGeometryBuilder** to ensure that a **JoobGeometry** can be built only if valid call sequences are issued. For example, calling **BeginFigure** without first calling **BeginGeometry** or when creating a polygon by calling **EndFigure** without issuing at least three calls to **AddLine** violate the state machine and result in an exception.

You can also use the **JoobGeometryBuilder** to create composite geometries such as **MultiPoint** and **MultiLineString**, or even nested composite geometries; for example, one **GeometryCollection** nested inside another **GeometryCollection**, as shown in the following example.

```
public JoobGeometry CreateNestedGeometryCollection()
{
    var builder = new JoobGeometryBuilder();
    builder.BeginGeometry(GeometryType.GeometryCollection);
    {
        builder.BeginGeometry(GeometryType.Point);
        builder.BeginFigure(1, 2);
        builder.EndFigure();
        builder.EndGeometry();

        builder.BeginGeometry(GeometryType.GeometryCollection);
        {
            builder.BeginGeometry(GeometryType.Polygon);
            builder.BeginFigure(1, 2);
            builder.AddLine(3, 4);
            builder.AddLine(4, 1);
            builder.AddLine(1, 2);
            builder.EndFigure();
            builder.EndGeometry();

            builder.BeginGeometry(GeometryType.LinearRing);
            builder.BeginFigure(2, 8);
            builder.AddLine(4, 2);
            builder.AddLine(4, 7);
            builder.AddLine(2, 8);
            builder.EndFigure();
            builder.EndGeometry();

            builder.BeginGeometry(GeometryType.MultiPolygon);
            {
                builder.BeginGeometry(GeometryType.Polygon);
                builder.BeginFigure(1, 2);
                builder.AddLine(3, 4);
                builder.AddLine(4, 1);
                builder.AddLine(1, 2);
                builder.EndFigure();
                builder.EndGeometry();

                builder.BeginGeometry(GeometryType.Polygon);
                builder.BeginFigure(2, 8);
                builder.AddLine(4, 2);
                builder.AddLine(4, 7);
                builder.AddLine(5, 10);
                builder.AddLine(2, 8);
                builder.EndFigure();
                builder.EndGeometry();
            }
            builder.EndGeometry();
        }
        builder.EndGeometry();
    }
    builder.EndGeometry();
```

```
        return builder.GetConstructedGeometry(ClassPersistence.Persistent);
    }
```

Querying and Analyzing Geometries

As mentioned earlier in this appendix, **JoobGeometry** provides a number of properties and methods for information inquiry and spatial analysis. Most of these properties and methods are self-explanatory and easy to use.

For more details, see the JADE .NET API documentation (that is, the **JadeDotNetAPI.chm** file in the installed JADE **documentation** directory; for example, **C:\Jade\JADE Docs\documentation**).

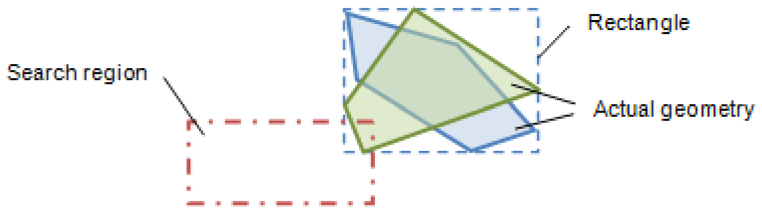
Spatial Index

Spatial indexes are used to organize a collection of spatial objects in the ways that spatial queries can be optimized. There are many spatial index methods available; for example, **Quadtree**, **R-tree**, **X-tree**, and **kd-tree**. In JADE .NET, **R-tree** is chosen as the spatial index method.

R-tree is a tree data structure similar to **B-tree**, but is used for indexing multi-dimensional information. The **R** in **R-tree** stands for *rectangle*, indicating the shape **R-tree** uses to split space. For more details about **R-tree** and how it works, see the following Web sites.

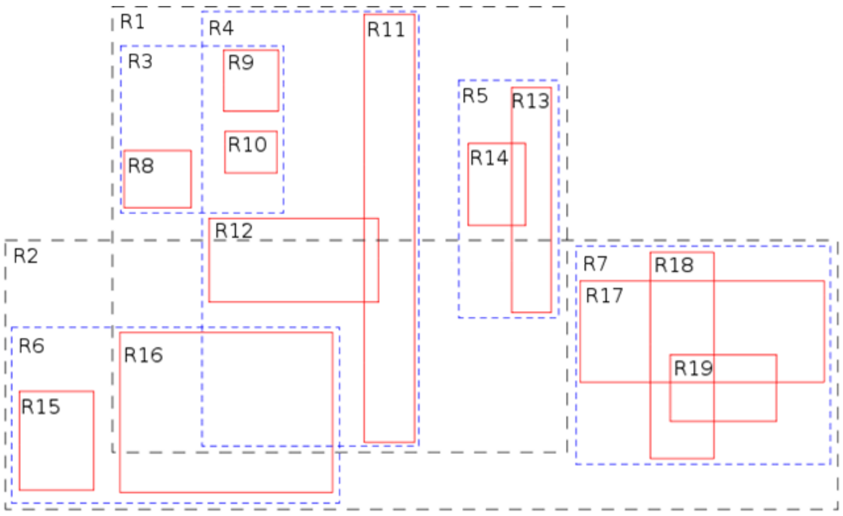
- R-tree on Wikipedia
- R-tree Portal

Since **R-trees** deal only with rectangles and different geometries may have the same bounding rectangle, searching an **R-tree** cannot always give you an accurate answer. For example, if you are looking for geometries in a **R-tree** that are intersecting with the search region in the following diagram, you end up with two geometries: both the blue one and the green one.

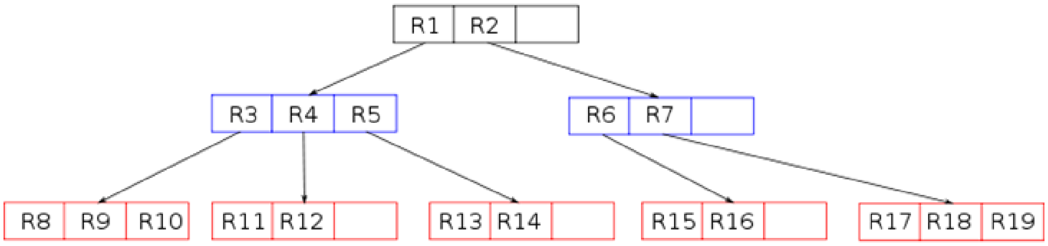


To obtain the accurate answer, you then need a second level of filtering concerning only the actual geometries, not their bounding rectangles.

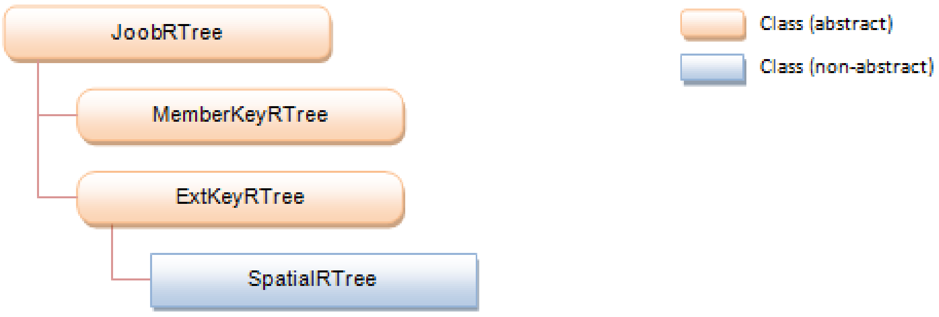
However, this second-level filtering could be a much more computational-intensive process, especially when the actual shape of a geometry is complex. Therefore, use **R-tree** to narrow down the number of geometries before passing them to the second-level filter.



When you are interacting with JADE .NET's **R-tree** APIs, you do not have to worry about this two-level filtering process, as it is automatically handled for you.



JADE .NET provides the following classes that enable you to work with R-trees.



JoobRTree is an abstract class encapsulating the implementation of the **R-tree** algorithm in JADE .NET. It exposes a series of search methods that enable you to find required **IMultiDimensionalObjects**. Be aware that **R-tree** can handle multi-dimensional data that may or may not reside in the spatial domain. **JoobRTree**, as well as **MemberKeyRTree** and **ExtKeyRTree**, are therefore generic types and expect you to specify the type parameters when you create subclasses.

In addition to the normal collection methods such as **Add** and **Remove**, **JoobRTree** contains a list of search methods for searching geometries spatially. Currently, JADE .NET supports eight spatial relationships: **contain**, **within**, **disjoint**, **intersect**, **touch**, **cross**, **overlap**, and **equal**.

To perform a spatial search, you can call the corresponding search method such as **ContainSearch** or **IntersectSearch**, or you can call the generic **Search** method with a relationship parameter; for example:

```
var rtree = // new ...

rtree.Add(new SpatialObject { Geometry = new JoobGeometry("...") });
rtree.Add(new SpatialObject { Geometry = new JoobGeometry("...") });
...
var searchRegion = new JoobGeometry("POLYGON ((...))");
var results = rtree.IntersectSearch(searchRegion);
// or
results = rtree.Search(searchRegion, MultiDimensionalObjectRelationship.Intersect);
```

MemberKeyRTree is an abstract subclass of **JoobRTree**. Similar to a **MemberKeyDictionary**, objects stored in a **MemberKeyRTree** are keyed using their own properties. However, there can be a single key only for a **MemberKeyRTree**, and the key must be of type **IMultiDimensionalObject**.

ExtKeyRTree is also an abstract subclass of **JoobRTree**. Each **ExtKeyRTree** can also have only a single key of type **IMultiDimensionalObject**.

SpatialRTree is a concrete subclass of **ExtKeyRTree**, with a key type predefined to **JoobGeometry**.

When you create a subclass from **JoobRTree** or one of its subclasses, you automatically benefit from all of the advantages provided by the JADE collection implementation; for example, inverse maintenance and distributed processing.