



# J A D E <sup>TM</sup>

## JADE Object Management and Persistence in Java

Jade Software Corporation Limited cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages, or loss of profits. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Jade Software Corporation Limited.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Copyright © 2008 Jade Software Corporation Limited.

All rights reserved.

JADE is a trademark of Jade Software Corporation Limited. All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

---

# Contents

<b>JADE Object Management and Persistence in Java</b>	<b>4</b>	
Introduction .....	4	
Java API Architecture.....	4	
Object Databases.....	5	
Developing from Java .....	6	
Java Code for JADE Access .....	6	
Example Code .....	6	
Java Inheritance Modes.....	8	
Java Class Fields .....	8	
Annotated Java Class.....	9	
Example .....	9	
Annotated References .....	10	
Collection Class Definition.....	11	
Array Collection .....	11	
Dictionary Collection .....	11	
EntityManagerFactory .....	12	
EntityAccess, EntityManager .....	12	
EntityTransaction .....	12	
Use in a Java Application Server .....	12	
Generation of a JADE Schema .....	13	
Deployment and Loading of the Schema.....	14	
Handling Updates following Java System Development .....	15	
Developing from JADE .....	15	
Java Exposure Definition .....	15	
Use of Class Text .....	15	
Java Exposure Generation .....	15	
Integration into Java Projects .....	15	
Summary.....	16	
<b>Appendix A</b>	<b>Example of Developing from Java</b>	<b>17</b>
<b>Appendix B</b>	<b>Example of Developing from JADE</b>	<b>28</b>
<b>Appendix C</b>	<b>Differences between JADE and Java</b>	<b>33</b>
Semantic Differences.....		33
Query.....		33
Relationships.....		33
Inverses .....		33
Relationship One-to-One, Uni-directional .....		33
Java Handling .....		34
JADE Handling .....		34

---

<b>Appendix C</b>	<b>Differences between JADE and Java, continued</b>	
	Relationship One-to-One, Bi-directional .....	34
	Java Handling .....	34
	JADE Handling .....	35
	Relationship One-to-Many, Uni-directional .....	35
	Java Handling .....	36
	JADE Handling .....	36
	Relationship One-to-Many, Bi-directional .....	36
	Java Handling .....	37
	JADE Handling .....	37
	Fetch Depth and Lazy Instantiation .....	38
	Cascading .....	39
	Object Lifetime .....	39
	Object Deletes .....	39
	Object Identity .....	39
	Detached Objects .....	40
	Exceptions .....	40
	Lock Exceptions .....	40
	Run Time Handling .....	41
	Transaction State and Persisting .....	41
	Rollback .....	41
	Locking .....	41
	Optimistic Locking .....	42
	Caching .....	42
	Notifications .....	43
	Database Connections .....	43
	Recovery from Persistence Provider Exceptions .....	44

---

# JADE Object Management and Persistence in Java

---

## Introduction

JADE Object Management and Persistence in Java (the **Java API**) provides a mechanism to manage Java objects (both persistent and transient) using the JADE object manager, and to directly persist Java objects using the object-oriented JADE database. It allows for full transaction control, multi-user object access, concurrency control with object-level locking, publish-and-subscribe events, and automatic object caching across distributed application servers. Through the Java API, all of the capabilities of the JADE object manager become available to Java applications.

The Java API is oriented to the following audiences.

- New customers and partners with established Java investments.  
The feature allows management and persistence of Java objects using JADE in a natural manner. JADE has full Object-Oriented Database Management System (OODBMS) capabilities.
- Existing or new customers with predominantly JADE-based systems.  
The feature allows a Java front-end to be added to existing JADE functionality, managing both persistent and transient objects.

## Java API Architecture

The Java API makes all JADE object management services available to Java.

For both persistent and transient objects, basic object create, read, update, and delete operations are supported. Java objects are created as regular instances, accessed and updated via standard 'get' and 'set' methods, and persisted or deleted with a single method call. Objects are read by simply following property references.

Object modeling is supported by relationships between classes and objects. Relationships are expressed by one-to-one, one-to-many, and many-to-many references that can be automatically maintained.

Various lock options control multi-user access to persisted Java objects. Updating is done with full transaction control, and notifications allow automatic communication of changes of state between different parts of a Java system.

The Java API uses a natural interface for Java developers. Annotations are used to decorate Java code for JADE management and persistence options. No external configuration files are required for this.

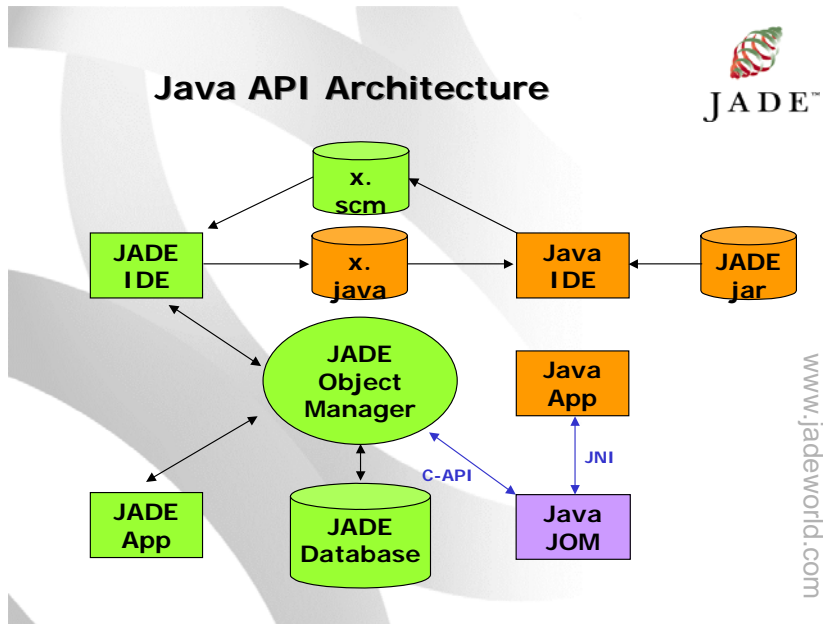
Access to JADE functionality is transparent.

JADE environment support provides access to existing JADE applications.

JADE methods can be called from Java, using both primitive and object parameters.

Environmental classes in the JADE RootSchema such as **System**, **Node**, **Process**, **Application**, and **Global** can be used, and JADE environmental objects such as **system**, **node**, and **process** can be read.

For more details about JADE functionality, see [www.jadeworld.com/jade/overview.htm](http://www.jadeworld.com/jade/overview.htm).



Development from Java is done using the JADE **javajom.jar** library, which contains the interface classes for object management and persistence for the Java application.

Development from JADE is done using the JADE development environment (IDE) and generating Java source files for use in a Java application.

Runtime integration between Java and JADE is done using both the **javajom.jar** library and the **javajom.dll** to interact with the JADE object manager.

## Object Databases

Object databases, including JADE, provide the following benefits to application developers. Each of these applies to using a JADE database for Java object persistence.

- Composite objects and relationships. Objects can store references to other objects as deeply nested as required, to provide one composite entity via references. In an RDBMS, this is done by a single table with many null fields or a number of smaller tables linked using foreign keys, in which case a number of joins are required to gather the data. An object model can handle more-complex interrelated data than relational tuples. A related advantage is handling class hierarchies, which in an RDBMS is done with separate tables or with a single table with a type identifier field and nullable fields for the subclass properties.

- Impedance mismatch. A significant amount of developer and application time can be spent mapping objects to tables and back, which is completely avoided using an object database. There is also one data model only, rather than separate RDBMS tables and application objects.
- No query language is required to access data, which is done by simply following object references.
- No unique primary keys are required, as the object database automatically maintains object identifiers (oids). There is no limitation on the values that can be stored in an object.

Application development using an object database such as JADE can be faster than using a relational database, because of the natural fit with a complex data model, the natural reference navigation, and the lack of database mapping. Run time performance can also be faster, depending on the structure of the data. A JADE database is ACID-compliant and provides the full auditing and recovery functionality of a typical RDBMS.

JADE takes advantage of its single data model to handle schema changes and database reorganizations automatically, often in a way that is more seamless than relational databases. In addition to the Java API, JADE provides C and C++ APIs (a C# API will be available in 2008), Web services, as well as both relational query and relational export capabilities, so that persistent data can be accessed from other technologies. JADE's Relational Population Services (RPS) can replicate data automatically to relational databases in near real-time, enabling JADE applications to interoperate with relational databases for reporting, business intelligence, and data warehousing purposes, if required.

## Developing from Java

This section covers developing JADE applications from Java.

For an example of developing a Java application that uses JADE's object management and persistence, see Appendix A.

## Java Code for JADE Access

This section describes using Java code to access JADE.

### Example Code

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory();
EntityManager em = emf.createEntityManager();
EntityTransaction et = em.getTransaction();

// create a new sales tender
et.begin();
Tender tender =
em.createObject(Tender.class, JadeConstants.ObjectLifetime.PERSISTENT);
tender.setPrice(price);
tender.setMyClient(client);
em.persist(tender);
et.commit();
```

The JADE Java API is modeled on the Java Persistence API (JPA) introduced in Java 5, but it is not an implementation of it. There are sufficient differences, as the Java Persistence API has capabilities that do not apply to JADE (for example, querying) and it does not have some capabilities that JADE has (for example, notifications and multi-keyed collections).

However, the JADE Java API uses classes with the same names as the core classes of the Java Persistence API, for equivalent functionality. It also uses annotations based on the Java Persistence API to mark classes and properties to be persisted. Additionally, the JADE Java API can be used to manage both persistence and transient objects, if required.

Class **com.jadeworld.jade.entitymanager.Persistence** is used in Java SE to create a **com.jadeworld.jade.entitymanager.EntityManagerFactory**, which encapsulates a connection to a JADE database and which is created by **Persistence.createEntityManagerFactory()**. All communication with JADE objects is done using **com.jadeworld.jade.entitymanager.EntityManager**. An instance of this is created by **EntityManagerFactory.createEntityManager()**. Transaction control is done using a **com.jadeworld.jade.entitymanager.EntityTransaction** instance, obtained by **EntityManager.getTransaction()**.

Classes to be used in interactions with a JADE system are annotated with **@Entity** for a regular class or **@CollectionEntity** for a collection class. Instances of these classes can be transient or persistent, used as method parameters in calls to existing JADE methods, or both. This allows for both persistent and transient instance usage.

Because both JADE and Java are object-oriented, there is a one-to-one mapping between a Java class and its corresponding persisted JADE class. JADE classes have the same inheritance and polymorphism capabilities as Java classes.

There is no need for any mapping that is used in object-to-relational systems, other than to allow for differences between JADE and Java class names, property names, and property types. These mappings are all done using class and field annotations.

Each managed Java object (persistent or transient) has a unique JADE object, and each existing JADE object that is read has a unique Java object, regardless of the way in which the read is done. The mapping between the objects is held internally for each **EntityManager** instance.

One external configuration file only is required for the JADE Java API. This is the **persistence.xml** file used for the Java Persistence API. The usage of this in JADE is very similar to that in Java. No other external XML deployment descriptor files are required.

Once annotated Java classes have been written, a JADE schema file can be generated from them. This schema can be loaded into a JADE development environment and a set of annotated Java classes generated from the JADE classes that correspond to the original Java classes. In this way, changes can be round-tripped between Java and JADE.

The feature is implemented via a C++ DLL coded using the Java Native Interface (JNI). This DLL is called from the **EntityManager** class methods and it accesses the running JADE system directly. The DLL is loaded when the **EntityManagerFactory** is created. It starts up a JADE node and then signs on to the JADE system.

The equivalent JADE object instances for the Java objects are accessed directly from, or updated directly to, the JADE cache using an existing JADE C++ interface to the JADE object manager. This makes the management and persistence of Java objects transparent and allows the full functionality of the JADE object manager to be automatically available to the Java system.

## Java Inheritance Modes

This section describes the modes of Java inheritance.

- Inherited.  
Inherits from **com.jadeworld.jade.rootschema.Object** and is oriented to existing JADE systems.
- (Plain Old Java Object) POJO-like.  
Has no inheritance requirements but requires an **Id** field in the class.

Either style can be used, and a single Java application can use a mix of styles.

The **Inherited** style is more direct, and applies when the Java classes are not in an existing hierarchy. Java applications using an existing JADE system and using Java classes generated from JADE would naturally fit into this pattern, although both styles of classes can be generated from JADE.

The **POJO** style applies when the Java classes are in an existing hierarchy and are required to be managed or persisted by JADE.

## Java Class Fields

This section describes Java class fields. It applies to both persistent and transient objects under JADE management.

- Normal Java fields  
Normal Java fields exist only in the Java object and not in the corresponding JADE object. Normal Java manipulation is done on these fields, which can be declared using all available Java types.
- DbProperties  
**DbProperty** fields exist only in the corresponding JADE object. The field values are accessed through 'get' and 'set' property method calls. The field can be defined using all available JADE types as the equivalent Java type.
- DbFields  
**DbField** fields exist in both the JADE and the Java object. Normal Java manipulation is done on these fields. **DbField** fields apply only to JADE primitive types.

Any Java class annotated with **@Entity** can have all three styles of field defined and all styles apply to both Inherited and POJO-style class definitions.

A normal Java field is annotated with **@Transient** and takes no direct part in the JADE management or persistence. These would typically be derived fields where the value is computed from the values of other fields, or fields used only in Java processing that is not affected by any persistence.

A field annotated with **@DbProperty** is a virtual property that exists only in the corresponding JADE object. It is defined in Java using a 'get' or 'is' method, named according to the JavaBeans convention, with an optional 'set' method. These methods are coded to use an **EntityAccess** class method to get and set the value by calling into JADE.

A field annotated with **@DbField** is a regular Java field, which is also required to have 'get' and 'set' methods defined according to the JavaBeans convention. Its value is automatically set from the corresponding JADE object on reading, and copied to the JADE object on updating or persisting.

The difference between the **DbField** and **DbProperty** styles is that a **DbProperty** value is always fetched from the JADE cache so it is always up-to-date with other changes made to the object (for more details about caching, see "Fetch Depth and Lazy Instantiation" and "Object Lifetime", later in this document), but the access is slower than a normal Java field access as it has to call into JADE and back. A **DbField** value is current at the time of creating the Java object from JADE but it then remains the same, regardless of any other changes made to the JADE object; it is in fact a Java cache of the value.

If the value is to be accessed many times in Java and is known not to change often, a **DbField** is faster than a **DbProperty**.

The **EntityManager refresh(obj)** method can be used to refresh the values of all **DbField** fields to the current JADE values and the **persist(object)** method can be used to update the JADE fields from the current Java values.

## Annotated Java Class

This section describes the annotated Java class.

### Example

```
import com.jadeworld.jade.persistence.*;
import com.jadeworld.jade.entitymanager.EntityAccess;

@Entity
public class Person
{
    @Transient
        public boolean status;           // normal Java field
    @Id
        public long idField;
    @DbProperty
        public int getCode() {           // only in JADE
            return EntityAccess.getIntegerProperty(this, "code");
        }
        public void setCode(int code) {
            EntityAccess.setIntegerProperty(this, "code", code);
        }
    @DbField(length=30)
        public String name;             // in JADE and Java
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
}
```

The annotated Java class is a POJO-style class. It does not inherit from any JADE Java API class and has the required identifier field, named **idField** (any name can be used), of type **long** or **Long**, and annotated with **@Id**.

The **status** field is a Java-only field, annotated with **@Transient** to indicate this.

The **code** field is a virtual field whose value is obtained from JADE and set to JADE each time the **getCode()** or **setCode()** method is called. The call to JADE uses static methods on the **EntityAccess** class.

The **name** field value is set from JADE when the Java object is first created from a JADE object, or set to JADE when the Java object is updated or persisted.

## Annotated References

This section describes annotated references. The collection definition in class **Agent** is as follows.

```
@OneToMany(relationshipType=ReferenceRelationshipType.PEER,updateMode=ReferenceUpdateMode.AUTOMATIC,inverse="myAgent")
public SaleItemByCategoryCodeDict<SaleItem> getAllSaleItems()
{
    return(SaleItemByCategoryCodeDict<SaleItem>
        EntityAccess.getReferenceProperty(this, "allSaleItems"));
}
```

Because JADE is fully object-oriented, all objects are found by following references from other objects. JADE objects all have a unique oid, or object identifier, which is assigned when the object is created and stays with the object for its lifetime, which may be many years. This value is used for object identity whenever the object is used, both in the underlying database and in memory, and for both transient instances and persistent instances.

When persisted, this value is equivalent to a relational database primary key.

The **com.jadeworld.jade.rootschema.Object getOidString()** method can access the oid value for any object if the current Java object inherits from this class or by an **EntityManager** instance **getOid(object)** method.

JADE classes are linked by references between them, which echo the one-to-one and the one-to-many relationships in the external user system that they model. Access through the system is navigational, following such links.

Typically, a root object holds the top-level references and collections. JADE makes it easier to define and find global singleton objects using the **EntityManager.firstInstance** method, which finds the first persisted instance of a specific class. This works across Java VMs and JADE nodes, as the JADE database is common across multiple JADE nodes. This navigation makes references very important in JADE, which can directly model all combinations of one and many relationships.

JADE has extended functionality in its collections over that of Java, in particular the ability to define dictionaries using multiple keys.

In Java, collections to be managed by JADE are defined as separate classes inheriting from **com.jadeworld.jade.rootschema.Collection**, using annotations to define options and key definitions, and generics to define the membership of the collection.

The properties of the model classes that define the model relationships are coded using the appropriate one of the **@OneToOne**, **@OneToMany**, **@ManyToOne**, and **@ManyToMany** annotations.

## Collection Class Definition

This section describes the collection class definition.

### Array Collection

```
import com.jadeworld.jade.rootschema.ObjectArray;

@CollectionEntity
public class CustomerArray<Customer> extends ObjectArray<Customer>
{
}
```

JADE collections are strongly typed in that individual classes with a declared membership class are defined for each, and the JADE object manager enforces that only instances of this member class or a subclass can be added to the collection.

The Java equivalent is a definition using generics, as shown in the above code example.

The **com.jadeworld.jade.rootschema.Collection** class, which is the superclass of all JADE collections, is defined using generics for the member class. As this class also implements the **java.util.Collection** interface, it can be used in Java code as a regular Java collection.

The iterator of this class is a **com.jadeworld.jade.rootschema.Iterator**, which implements the **java.util.Iterator** interface. This allows Java collection classes defined from JADE collection classes to be used in Java **for** loops, as shown in the following example.

```
// populate the country combo box
cmbCountry.addItem("-- Select --");
for (Country country : company.getAllCountries())
{
    cmbCountry.addItem(country);
}
```

Primitive arrays are also implemented in this way, but do not need to be individually defined as separate collection classes. Primitive arrays are instances of **com.jadeworld.jade.rootschema.Array** and are accessed in the same way as other references; for example:

```
@DbProperty()
public Array<String> getStringArray()
{
    return (Array<String>)
        EntityAccess.getReferenceProperty(this, "stringArray");
}
```

### Dictionary Collection

```
import com.jadeworld.jade.rootschema.MemberKeyDictionary;

@CollectionEntity(keys={@DictionaryKey(key=" codePrefix"),
    @DictionaryKey(key=" codeNumber")})
public class SaleByItemDict<Sale> extends MemberKeyDictionary<Sale>
{
}
```

JADE has a collection type of **MemberKeyDictionary**, which has one or more keys that are properties of the member class. The keys can be of any type and each one is ascending or descending. This class definition is of a dictionary collection with membership **Sale** and keys of **codePrefix** and **codeNumber**, which are properties of the **Sale** class. These key properties can be of type **DbField** or **DbProperty**, but not **Transient**, as they must exist in the corresponding JADE object.

## EntityManagerFactory

The **EntityManagerFactory** is the top-level interface between Java and JADE.

## EntityAccess, EntityManager

All object interactions between Java and JADE are done using methods called from static methods on **EntityAccess** or from an instance of **EntityManager**.

## EntityTransaction

An **EntityTransaction** instance is obtained from the **EntityManager** instance and is used for transaction control.

## Use in a Java Application Server

So that the JADE API can be used in a Java application server, the JADE:

- **EntityManagerFactory** class implements the **javax.persistence.EntityManager** interface
- **EntityManager** class implements the **javax.persistence.EntityManager** interface
- **EntityTransaction** class implements the **javax.persistence.EntityTransaction** interface

This implementation is not a functional implementation, as several **EntityManager** methods have an empty implementation (for example, those related to RDMS queries) and the methods that are implemented do not necessarily function in exactly the same way as a Java Persistence API implementation.

This allows the **java.persistence.PersistenceUnit** and **PersistenceContext** annotations to be used on a definition of a **javax.persistence.EntityManagerFactory** and **EntityManager**, respectively; for example:

```
@PersistenceUnit(unitName="PortSchemaPU")
private javax.persistence.EntityManagerFactory emf;

@PersistenceContext(unitName="PortSchemaPU")
private javax.persistence.EntityManager em;
```

This code can be used in any type of Java EE-managed container (for example, a servlet or Entity Bean) in a Java application server. The application server locates the **persistence.xml** file for this application at server startup and application deploy time, and creates an **EntityManagerFactory** instance using the properties in this file. This starts the JADE node and signs on to it. This **EntityManagerFactory** instance is kept by the Java application server and is used when a Java EE container with one of the above annotations is created. It injects this instance into the annotated property, directly for an **EntityManagerFactory** and via a **createEntityManager** call for an **EntityManager**.

The **EntityManager** instance created this way is actually of an application server-specific class, so a **getDelegate()** method call is needed to get the JADE **EntityManager** instance. The JADE instance is the one that must be used to get the JADE functionality.

The application server also automatically calls the **EntityManagerFactory close()** method when the application is redeployed and when the application server closes down.

The **javajom.jar** library must be copied to the appropriate third-party library directory of the application server.

The application server can also require the **persistence.xml** file to have entries for **<jta-data-source>** and **<non-jta-data-source>**, even though they are not used by JADE. Use the default data sources that come with the particular application server.

An alternative way to use the JADE API in Java EE is the same as that for Java SE, where the **EntityManagerFactory** is explicitly created from the **Persistence.createEntityManagerFactory()** method in some application startup code and the factory instance carried between the classes, as required. The factory **close()** method must also be called from the appropriate closedown code.

## Generation of a JADE Schema

Each JADE system is implemented as a *schema*, which is the highest-level organizational structure in JADE and represents the object model for a specific domain. A schema is a logical grouping of classes, together with their associated methods and properties. These effectively define the object model upon which the applications are based.

JADE schemas can be defined in a hierarchy; for example, a superschema with the model classes and a subschema with the view classes.

Each JADE schema also contains one or more JADE *applications*, which is an end-user runtime system, typically defining a collection of forms and other runtime information such as the start-up form and the location of the help file. Applications can also be non-GUI, and JADE systems generated from Java classes have an application of this type. At least one JADE application is required for each schema. When a JADE system is started, a specified JADE application is run for a specified JADE schema.

Java packages are used to map to JADE schemas. All annotated classes in a Java package become a JADE schema whose name is specified by the **name** attribute of the **@Schema** annotation in the **package-info.java** file within the package. The JADE application name is specified by the **applicationName** attribute.

JADE systems are usually developed using the JADE development environment, which holds JADE class definitions and method source in a JADE database, so there are no separate class source files as is typical for Java. However, when a JADE system is deployed, the schema is extracted to two files. One file holds the class details and the other holds JADE form and application details. These files are loaded into the target JADE environment to create a new JADE system for this schema. This includes any data reorganization that may be required from new persistent class properties, deleted properties, and changes of existing properties. Any such reorganization can be automatically done as part of the schema load.

When a Java system is ready to be deployed to JADE, one or more JADE schema files must be generated from the Java class definitions and loaded into the appropriate JADE environment. This is done using a utility in the **javajom.jar** file, implemented in class **com.jadeworld.jade.tool.AnnotationProcessor**. As this is the **Main-Class** in **javajom.jar**, it can be run as:

```
java -jar javajom.jar <parameters>
```

The runtime parameters that define the location and definition of the Java system are:

- **-userClasses <directory>**

This parameter specifies the location of the user-annotated Java classes. These can be in **.class** form or packaged in a **.jar** file.

**-uc** is a synonym for **-userClasses**.

- **-persistenceUnitName <name>**

This parameter specifies the name of the **<persistence-unit>** entry in the **persistence.xml** file for this system. This is used to get the list of class names, and must have the correct value; otherwise no classes will be found from which to create the JADE schema.

**-pun** is a synonym for **-persistenceUnitName**.

A JADE schema file and forms file are written as **<JADE-schema-name>.scm** and **<JADE-schema-name>.ddb** for each JADE schema, corresponding to each Java package containing managed transient or persistent classes, to the directory specified by the first runtime parameter.

The schema file holds all JADE classes corresponding to the annotated Java classes and the forms file holds a JADE non-GUI application, which is run when the Java system signs on to the JADE system for this schema.

## Deployment and Loading of the Schema

Schemas can be loaded into JADE from the JADE development environment or via a batch process. For details, see Chapter 11 of the *JADE Development User's Guide* and Chapter 1 of the *JADE Administration Guide*.

The batch process uses **jadloadb.exe** in the JADE environment **bin** directory, where the main runtime parameters are:

```
path=database-directory
ddbFile=forms-file-name.ddb
schemaFile=schema-file-name.scm
ini=initialization-file-name
loadStyle=latestSchemaVersion
```

The following example shows the **jadloadb** command line for a JADE environment installed into **C:\Jade62** with the standard directories and names and a Java system at **C:\Java\classes** with a generated JADE schema in this directory of **JavaSchema**.

```
C:\Jade62\bin\jadloadb path=C:\Jade62\system
ini=C:\Jade62\system\jade.ini schemaFile=C:\Java\classes\JavaSchema.scm
ddbFile=C:\Java\classes\JavaSchema.ddb loadStyle=latestSchemaVersion
```

This example loads the schema as the latest version and initiates any data reorganization that is required following the schema load.

## Handling Updates following Java System Development

The steps described in the previous sections apply equally to a new JADE schema generated from Java and for subsequent generates as the Java system is developed.

The JADE schema load automatically updates the previous JADE class definitions and does any data reorganization that is required from persistence class changes.

## Developing from JADE

This section covers developing a Java application from JADE. For an example of developing a Java application from JADE, see Appendix B.

### Java Exposure Definition

The Java exposure allows for the selection of the JADE classes (both persistent and transient) that are to be exposed to Java.

This exposure is done by generating a **persistence.xml** file, a **package-info.java** file, and an annotated Java class for each selected JADE class. These classes provide a proxy Java class to the JADE class, with a Java property for each selected JADE property with the required access code and annotations for both the run time processing and generation of a JADE schema for round-tripping, if required.

### Use of Class Text

Additional Java code can be written in the text of exposed classes, delimited by **@Java** and an optional **@EndJava**.

This text is automatically added to the Java class file generated from this class. For examples, see the Erewhon example **ErewhonInvestmentsModelSchema** schema **Agent** and **Sale** classes.

### Java Exposure Generation

The **Generate Java** tab of the **Java Exposure Wizard** form is used to generate the Java entity classes from the exposed JADE classes and properties. The **Output directory for Java classes** text box value specifies where these files are written. A **persistence.xml** file is written to this directory, with **<class>** entries from the exposed class list and **<property>** entries from the remaining fields of this form.

Each exposed JADE class is written as an annotated Java class to a subdirectory whose name is the JADE schema name, with a package name of this schema name. Each such subdirectory also contains a **package-info.java** file with the schema annotation.

### Integration into Java Projects

The generated Java files must be copied into the source directory of the target Java application. The **persistence.xml** file goes into the **META-INF** directory at the top of the source directory hierarchy and the other files as Java packages using the existing names.

If the package names of the entity class files are changed, both the **package** line in the class file source and the **<class>** entry in the **persistence.xml** file must be changed to match.

## Summary

The Java API allows Java applications to use JADE to manage both transient and persistent objects, and provides a persistence back-end for Java applications to easily store persistent objects.

All of the run time and distributed processing capabilities of JADE are available to Java applications, without requiring the use of the JADE language at development time.

The object model for the persisted classes still needs to be designed taking into account the capabilities of the JADE object manager.

The Java API also allows existing JADE systems to be accessed from Java.

Much development is done in mixed environments. This feature allows both JADE and Java to be used together where each is most appropriate, and to integrate existing JADE systems with existing Java systems.

---

# Appendix A

# Example of Developing from Java

---

The example in this appendix assumes locations for both the Java classes and the JADE environment of **C:\Java\<project>\classes** for the Java application and **C:\Jade62** for the JADE environment, with the standard folders within this of **bin** and **system** and a JADE configuration file called **jade.ini** in the **system** folder. For Linux, replace these with the equivalent; for example, **/home/<user-name>/jade62** and ensure that the JADE environment is set. Modify the locations where they are used in this appendix to match the actual locations of your Java and JADE environments.

This example has a class of **Person**, each instance of which has a single **Address**. The **Person** class has a **name** property and the reference to the **Address** class, and **Address** has a **street** and **city** property. There is no link (inverse reference) from **Address** back to **Person**.

## ➤ To develop a JADE application from Java

1. To add a new project:
  - a. Create a new project of type Java application. A name of **JavaTest** is assumed in the examples in following steps of this instruction.
  - b. Add **javajom.jar** from **C:\Jade62\bin** (located in the **lib** directory on Linux) as a dependent library.
2. To define entity classes:
  - a. Create a new Java package named **javaTest**.
  - b. Create a new file named **package-info.java** in the Java package, with content as follows.

```
@Schema(name="JavaTestSchema", applicationName=
        "JavaTestApplication", defaultMapFile="javaMap")

package javaTest;

import com.jadeworld.jade.persistence.*;
```

This defines the JADE schema-level information.

- c. Create a new class named **Address** in package **javaTest**, as follows.

```
package javaTest;
import com.jadeworld.jade.persistence.*;
import com.jadeworld.jade.entitymanager.EntityAccess;

@Entity
public class Address
{
    public Address()
    {
    }
}
```

```
@Id
private long id;

@DbField
private String street;

public String getStreet() {
    return street;
}
public void setStreet(String street) {
    this.street = street;
}
@DbField
private String city;

public String getCity() {
    return city;
}
public void setCity(String city) {
    this.city = city;
}
}
```

This defines an annotated persisted entity class of **Address**.

- d. Create a new Java class named **Person** in the **javaTest** package, as follows.

```
package javaTest;
import com.jadeworld.jade.persistence.*;
import com.jadeworld.jade.entitymanager.EntityAccess;

@Entity
public class Person
{

    public Person()
    {
    }
    @Id
    private long id;

    @DbField
    private String name;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}

@DbProperty
@OneToOne()
public Address getAddress() {
    return (Address)
    EntityAccess.getReferenceProperty(this, "address");
}
```

```

        public void setAddress(Address address) {
            EntityAccess.setReferenceProperty(this, "address",
                address);
        }
    }
}

```

This defines an annotated persisted entity class of **Person**.

- e. Create a new class named **JavaTestApplication** in package **javaTest**, as follows.

```

package javaTest;
import com.jadeworld.jade.persistence.*;
import com.jadeworld.jade.rootschema.Application;

@ApplicationEntity
public class JavaTestApplication extends Application
{
}

```

Each JADE system requires a subclass of **Application**. The JADE schema write process adds a default subclass to the generated schema if you do not explicitly define it.

- f. Create a new class named **JavaTestGlobal** in package **javaTest**, as follows.

```

package javaTest;
import com.jadeworld.jade.persistence.*;
import com.jadeworld.jade.rootschema.Global;

@GlobalEntity
public class JavaTestGlobal extends Global
{
}

```

Each JADE system requires a subclass of **Global**. The JADE schema write process adds a default subclass to the generated schema if you do not explicitly define it.

3. To create a **persistence.xml** file:

- a. Create a new folder called **META-INF** in the source directory and create a new empty file called **persistence.xml** within this folder.
- b. Add the following text to the **persistence.xml** file and change the properties to match the actual installed locations of the JADE environment, as follows.

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
version="1.0" >
    <persistence-unit name="JavaTestPU" >
        <class>javaTest.Person</class>
        <class>javaTest.Address</class>
        <class>javaTest.JavaTestApplication</class>
        <class>javaTest.JavaTestGlobal</class>
    <properties>
        <property name="jade.database.path" value="C:/Jade62/system"/>
        <property name="jade.database.configuration"
            value="C:/Jade62/system/jade.ini"/>
        <property name="jade.database.multiuser" value="true"/>
        <property name="jade.database.username" value="user"/>
        <property name="jade.database.password" value="" />
    </properties>
</persistence-unit>
</persistence>

```

```

        <property name="jade.database.schema" value="JavaTestSchema"/>
        <property name="jade.database.application"
            value="JavaTestApplication"/>
    </properties>
    </persistence-unit>
</persistence>

```

Note that of the property values:

- **jade.database.path** is the system directory of the JADE environment
- **jade.database.configuration** is the JADE configuration file path
- **jade.database.multiuser** is **true** if there is already a JADE database server running (typically the case for a production or shared development system) and **false** if the JADE environment is not already running and you want to run it just for this application (typically the case for a local test development system)
- **jade.database.username** is any convenient name, although it should be one that is not already signed on to that environment if the JADE environment is already running
- **jade.database.password** is required only if the JADE environment enforces password entry, which is not the default case
- For the **@Schema** annotation in the **package-info.java** file in the **javaTest** package, **jade.database.schema** is the same as the **@Schema** annotation **name** attribute value
- For the **@Schema** annotation in the **package-info.java** file in the **javaTest** package, **jade.database.application** is the same as the **@Schema** annotation **applicationName** value

This **persistence.xml** file is the only external XML configuration file required for JADE object management and persistence. All classes to be managed or persisted by JADE must be named with a **<class>** entry; otherwise they will not be used at run time nor will JADE classes be created from them when a JADE schema is extracted. The **<property>** entries can be overwritten at run time when the **EntityManagerFactory** is created. The entries in this example are the default values to be used.

4. To create access and update code, create a new package named **main** and in this package, create a new **Main** class, as follows.

```

package main;

import com.jadeworld.jade.entitymanager.*;
import com.jadeworld.jade.JadeConstants;
import javaTest.Address;
import javaTest.Person;

public class Main
{
    EntityManagerFactory emf;
    EntityManager em;

    public Main()
    {
    }
}

```

```
public void init()
{
    emf = Persistence.createEntityManagerFactory("JavaTestPU");
    em = emf.createEntityManager();
}
public void closeDown()
{
    em.close();
    emf.close();
}
public void populate()
{
    // create two people with a shared address
    EntityTransaction et = em.getTransaction();
    et.begin();
    try {
        Address address1 =
            (Address)em.createObject(Address.class,
                JadeConstants.ObjectLifetime.PERSISTENT);
        address1.setStreet("123 High Street");
        address1.setCity("London");
        em.persist(address1);

        Person person1 = (Person)em.createObject(Person.class,
            JadeConstants.ObjectLifetime.PERSISTENT);
        person1.setName("Fred Jones");
        person1.setAddress(address1);
        em.persist(person1);

        Person person2 = (Person)em.createObject(Person.class,
            JadeConstants.ObjectLifetime.PERSISTENT);
        person2.setName("Bill Smith");
        person2.setAddress(address1);
        em.persist(person2);
        et.commit();
    } catch (RuntimeException e) {
        e.printStackTrace();
        et.rollback();
    }
}
// read back and print the objects from JADE
public void read()
{
    for (Person person : em.instances(Person.class))
    {
        System.out.println("Person " + person.getName() + " " +
            person.getAddress().getStreet() + " " +
            person.getAddress().getCity());
    }
}

public static void main(String[] args)
{
    Main main = new Main();
    main.init();
    try {
        main.populate();
        main.read();
    } catch (RuntimeException ex) {
        ex.printStackTrace();
    }
}
```

```

        } finally {
            main.closeDown();
        }
    }
}

```

This code creates a single **Address** instance and two **Person** instances, with each **Person** object having the same shared address. The initialize code creates the **EntityManager** and the **EntityManagerFactory**.

The sign-on to JADE using the properties from the **persistence.xml** file is done when the **EntityManagerFactory** is created.

The objects are then persisted to JADE and the connection to JADE is closed. The main processing is done within a **try/catch** block, as any exceptions from the JADE processing are runtime exceptions. The object persistence is also done within a **try/catch** block, with a **rollback()** being done in the **catch** code to undo the database update if an exception is raised.

The **populate** method uses the **EntityManager** class **createObject** method to create a new Java object that has the associated JADE object created and the internal **id** property set to link the two. An alternative way is to create the Java object separately and then use the **EntityManager persist** method to create the JADE object and link the two.

The **read** method uses the **EntityManager instances** method to return a collection of all instances of the specified class. This is the functional equivalent of an SQL **SELECT \* FROM table**.

5. To create a JADE schema, build the project, if necessary, to create the Java class files or **.jar** file and then run the following from a command line.

```

java -jar C:\Jade62\bin\javajom.jar -uc <java-class-location>
-pun JavaTestPU

```

If you are using Eclipse with the default **Build automatically** option set, you do not need to do an explicit build step.

The **<java-class-location>** is the directory of the built Java class files or **.jar** file, depending on the Java development environment or build mechanism that you use.

For an Eclipse project with separate source and class file folders, the Java class location is **<workspace-directory>/JavaTest/bin** and for a NetBeans standard project, the Java class location is **<project-path>/JavaTest/build/classes** or **<project-path>/JavaTest/dist/Javatest.jar**.

This creates **JavaTestSchema.scm** and **JavaTestSchema.ddb** files in the folder specified in **<java-class-location>**.

6. To load the schema into JADE, run the JADE database server for this JADE environment and then run the following from a command line. If you are running the JADE environment in single user mode, replace **server=multiUser** with **server=singleUser** in the command, and make sure that the JADE environment is not already running.

```

C:\Jade62\bin\jadloadb path=C:\Jade62\system
ini=C:\Jade62\system\jade.ini
schemaFile=<java-class-location>\JavaTestSchema.scm
ddbFile=<java-class-location>\JavaTestSchema.ddb
loadStyle=latestSchemaVersion server=multiUser

```

This loads the JADE schema files into this JADE environment. The output should look similar to the following.

```
Opening database in C:\Jade62\system...
Database opened
Loading schema file C:\Java\classes\JavaTestSchema.scm...
Compile complete
Committing schema file changes...
Creating application and/or global instances...
Loading forms file C:\Java\classes\JavaTestSchema.ddb...
Rebuilding form build data...
Waiting for running applications...
Closing database...
Database closed
Load completed
```

7. To run this Java application, ensure that the JADE directory **C:\Jade62\bin** is in the system path (Windows **PATH** variable) and set the **java.library.path** Java system property to **C:\Jade62\bin**. If you run the application from the command line, add **-Djava.library.path=C:\Jade62\bin** to the execution string. If you run the application from a Java development environment, set the **VM arguments** property to **-Djava.library.path=C:\Jade62\bin**.

In Eclipse, this is on the **Arguments** tab of the Run configurations dialog. In NetBeans, this is on the **Run** category of the project properties dialog.

Run this Java application, which should print the following to the console.

```
Person Fred Jones lives at 123 High Street London
Person Bill Smith lives at 123 High Street London
```

---

**Note** If you run this again, it creates a second set of **Person** and **Address** objects and it print four lines, and so on, for each successive run.

---

The **Address** class is now changed to hold a dictionary collection by name of **Person** objects that live at this address, and both classes have an additional field, defined this time as type **DbProperty**.

8. Modify the entity classes and the access and update code to extend update and access code to use these additional properties and to read via the new collection, as follows.
  - a. Create a new Java class named **PersonNameDictionary** in package **javaTest**, as follows.

```
package javaTest;

import com.jadeworld.jade.rootschema.MemberKeyDictionary;
import com.jadeworld.jade.persistence.*;
import com.jadeworld.jade.entitymanager.EntityAccess;

@CollectionEntity(duplicatesAllowed=true,
    keys={@DictionaryKey(key="name")})
public class PersonNameDictionary<Person> extends
MemberKeyDictionary<Person>
{
}
```

**PersonNameDictionary** is a JADE **MemberKeyDictionary** subclass that inherits the JADE collection behavior but also implements the Java **java.util.Collection** interface. It can therefore be used as a regular Java collection. It allows direct access to member **Person** objects by their **name** property value.

- b. In class **Person**, change the code using the **address** property to the following.

```
@DbProperty()
@ManyToOne(relationshipType=ReferenceRelationshipType.CHILD,
    updateMode=ReferenceUpdateMode.MANUAL,inverses={"persons"})
public Address getAddress() {
    return (Address)
        EntityAccess.getReferenceProperty(this, "address");
}
public void setAddress(Address address) {
    EntityAccess.setReferenceProperty(this, "address", address);
}
```

- c. In class **Person**, add the new **dateOfBirth** property by using the following code.

```
import java.util.Calendar;

@DbProperty(type="Date")
public Calendar getDateOfBirth()
{
    return EntityAccess.getDateProperty(this, "dateOfBirth");
}
public void setDateOfBirth(Calendar date)
{
    EntityAccess.setDateProperty(this, "dateOfBirth", date);
}
```

- d. In class **Address**, add the following code.

```
@DbProperty
public String getAreaCode() {
    return EntityAccess.getStringProperty(this, "areaCode");
}
public void setAreaCode(String areaCode) {
    EntityAccess.setStringProperty(this, "areaCode", areaCode);
}

@DbProperty()
@OneToMany(relationshipType=ReferenceRelationshipType.PARENT,
    updateMode=ReferenceUpdateMode.AUTOMATIC,inverse="address")
public PersonNameDictionary<Person> getPersons() {
    return (PersonNameDictionary<Person>)
        EntityAccess.getReferenceProperty(this, "persons");
}
```

This code adds the **areaCode** property and sets up the relationship between **Address** and **Person** as one-to-many, implemented by the **persons** collection on **Address**, with an inverse of **address** on **Person**.

Because this address property is the manual side of the relationship (set by the **updateMode** property of the **@OneToMany** annotation), setting this property automatically also adds this **Person** object to the **Address persons** collection, as is shown in the **populate** method in step f of this instruction.

- e. Add the following import statements to the **Main** class.

```
import java.util.GregorianCalendar;
import java.util.Locale;
import java.text.DateFormat;
```

- f. Change the **Main** class **populate()** method to the following.

```
public void populate()
{
    // create two more people with a new shared address
    EntityTransaction et = em.getTransaction();
    et.begin();
    try {
        Address address1 = (Address)em.createObject(Address.class,
            JadeConstants.ObjectLifetime.PERSISTENT);
        em.persist(address1);
        address1.setStreet("456 High Street");
        address1.setCity("London");
        address1.setAreaCode("N1E2W3S4");
        em.persist(address1);

        Person person1 = (Person)em.createObject(Person.class,
            JadeConstants.ObjectLifetime.PERSISTENT);
        person1.setName("Jack Smith");
        person1.setDateOfBirth(new GregorianCalendar(1962, 3,
            21));
        person1.setAddress(address1);
        em.persist(person1);

        Person person2 = (Person)em.createObject(Person.class,
            JadeConstants.ObjectLifetime.PERSISTENT);
        person2.setName("Betty Smith");
        person2.setAddress(address1);
        person2.setDateOfBirth(new GregorianCalendar(1963, 2,
            12));
        em.persist(person2);
        et.commit();
    } catch (RuntimeException e) {
        e.printStackTrace();
        et.rollback();
    }
}
```

- g. Change the **Main** class **read()** method to the following.

```
public void read()
{
    DateFormat dateFormatter =
        DateFormat.getDateInstance(DateFormat.DEFAULT, new
        Locale("en", "NZ"));
    Address lastAddress=null;
    // Person to Address
    for (Person person : em.instances(Person.class))
    {
        String dob = "null";
        if (person.getDateOfBirth() != null)
            dob = dateFormatter.format(
                person.getDateOfBirth().getTime());
    }
}
```

```

        System.out.println("Person " + person.getName() +
            " born " + dob + " lives at " +
            person.getAddress().getStreet() + " " +
            person.getAddress().getCity() + " " +
            person.getAddress().getAreaCode());
    }
    // Address to Person
    for (Address address : em.instances(Address.class))
    {
        for (Person person : address.getPersons())
        {
            System.out.println("At address " +
                address.getStreet() + " " + address.getCity()
                + " lives " + person.getName());
            lastAddress = address;
        }
    }
    // direct access to Persons by name
    Person person = lastAddress.getPersons().getAtKey("Betty
Smith");
    System.out.println("Specific person from dictionary " +
        person.getName());
}

```

The **read** method now has a collection reference off the **Address** object being read using the Java **for** loop and an individual **Person** object being read from this dictionary collection using the **getAtKey** method.

- h. Add the following line to the **<class>** block of the **persistence.xml** file.

```
<class>javaTest.PersonNameDictionary</class>
```

9. To recreate the JADE schema, rebuild the Java application, if necessary, and run the following from a command line.

```
java -jar C:\Jade62\bin\javajom.jar -uc <java-class-location>
-pun JavaTestPU
```

For details about **<java-class-location>**, see step 5 of this instruction.

This creates new **JavaTestSchema.scm** and **JavaTestSchema.ddb** files in **C:\Java\JavaTest\classes**, overwriting the previous files with these names.

10. To reload the schema into JADE, run the following from a command line when the database server is still running, if you are running the JADE environment in multiuser mode.

If you are running the JADE environment in single user mode, replace **server=multiUser** with **server=singleUser** in the command and make sure that the JADE environment is not already running.

```
C:\Jade62\bin\jadloadb path=C:\Jade62\system
ini=C:\Jade62\system\jade.ini
schemaFile=<java-class-location>\JavaTestSchema.scm
ddbFile=<java-class-location>\JavaTestSchema.ddb
loadStyle=latestSchemaVersion server=multiUser
```

The output should look similar to the following.

```
Opening database in C:\Jade62\system...
Database opened
Loading schema file C:\Java\classes\JavaTestSchema.scm...
Compile complete
Committing schema file changes...
Loading forms file C:\Java\classes\JavaTestSchema.ddb...
Performing data reorganization...
Waiting for reorg to complete ...
Reorg succeeded
Rebuilding form build data...
Waiting for running applications...
Closing database...
Database closed
Load completed
```

11. Run the Java application, which should print the following to the console.

```
Person Fred Jones born null lives at 123 High Street London
Person Bill Smith born null lives at 123 High Street London
Person Jack Smith born 21/04/1962 lives at 456 High Street London
      N1E2W3S4
Person Betty Smith born 12/03/1963 lives at 456 High Street London
      N1E2W3S4
At address 123 High Street London lives Bill Smith
At address 123 High Street London lives Fred Jones
At address 456 High Street London lives Betty Smith
At address 456 High Street London lives Jack Smith
Specific person from dictionary Betty Smith
```

---

**Notes** The existing **Person** instances (**Fred Jones** and **Bill Smith**) have been automatically added into the **persons** collection of the existing **Address** instance by the data reorganization that followed the JADE schema load, using the value of their existing **address** property, which has now become an inverse of this new **persons** collection.

The existing **Person** and **Address** instances now have the new properties defined but with null or empty values (the **Date** field is null and the **String** is an empty string), as a result of the data reorganization following the second JADE schema load.

---

---

# Appendix B

# Example of Developing from JADE

---

The example in this appendix uses the **Erewhon** sample system, which you can load from the **examples\erewhon** directory of the JADE installation directory.

A new Java exposure is created and Java classes are generated for **Client**. Java code is then written to access and update the **Client** objects in the JADE database, using these generated Java classes.

To protect against simultaneous updates, the Java code to update a **Client** uses locking and a lock exception handler.

## ➤ To develop a Java application from JADE

1. To create a Java exposure:
  - a. In the **ErewhonInvestmentsViewSchema** schema, add a new Java exposure named **Client**.
  - b. Select the **Superschemas up to** option and then select a **top most Schema** of **ErewhonInvestmentsModelSchema**.
  - c. Set the **Default Java classes to** option to **Inherited** and the **Default primitive features to** option to **Property**.
  - d. Expose the **AddressableEntity**, **Client**, and **Company** classes as inherited.

Expose all properties of **AddressableEntity** as type **property**; that is, the **Field** check box on the right of the **Java Feature Mappings** sheet is not selected, but only **myCompany** of **Client** and **allClients** of **Company**.
  - e. Generate the Java classes to the required location.
2. In Java:
  - a. Create a new Java application project with the **Main** class in one package and a separate package named **ErewhonInvestmentsModelSchema**.

Alternatively, you can name this package to meet your requirements and modify the generated Java sources after copying them, with the **package** line changed to match.

Add **javajom.jar** from **C:\Jade62\bin** as a dependent library.
  - b. Copy in the generated source files into this package directory in the project source directory.
  - c. Create a new folder called **META-INF** in the source directory and copy the generated **persistence.xml** file into this folder.

Modify the **<class>** entries to correct the package names if you specified a name other than **ErewhonInvestmentsModelSchema**. Modify the **<property>** entries if the values need changing to sign on to the JADE system.

---

**Note** If you are using Eclipse, you may need to refresh the project to see the newly copied files. Do this by right-clicking on the **src** folder of the project and then selecting the **Refresh** option or by pressing the F5 key.

---

- d. Write a **Main** class, as follows. Change the package name on the first line as required to match your package name.

```
package main;

import com.jadeworld.jade.entitymanager.*;
import ErewhonInvestmentsModelSchema.Company;
import ErewhonInvestmentsModelSchema.Client;
import com.jadeworld.jade.JadeException;
import com.jadeworld.jade.LockExceptionHandler;
import com.jadeworld.jade.JadeConstants;

/**
 *
 * @author JADE Development Centre
 */
public class Main implements LockExceptionHandler
{
    EntityManagerFactory emf;
    EntityManager em;
    Company company;

    public void init()
    {
        emf = Persistence.createEntityManagerFactory(
            "ErewhonInvestmentsModelSchemaPU");
        em = emf.createEntityManager();

        company = em.firstInstance(Company.class);
    }
    public void closeDown()
    {
        em.close();
        emf.close();
    }

    // read and update a Client object
    private void update()
    {
        EntityTransaction et = em.getTransaction();
        et.begin();
        try {
            Client client = company.getAllClients().
                getAtKey("Roger Boeing");
            System.out.println("Client " + client.getName() +
                " " + client.getAddress1());
            // lock this object in preparation for updating it
            em.exclusiveLock(client, client);
            // toggle between two addresses so this can be run
            // repeatedly
            if (client.getAddress1().equals("167 Wingtip Road"))
                client.setAddress1("999 Wingtip Road");
            else
                client.setAddress1("167 Wingtip Road");

            et.commit();
        }
    }
}
```

```

        } catch (RuntimeException e) {
            e.printStackTrace();
            et.rollback();
        }
    }

    // Lock exception handler
    public JadeConstants.ExceptionAction
        exceptionRaised(JadeException exceptionDetails)
    {
        Object lockObject = exceptionDetails.getErrorObject();
        // retry 20 times then give up
        for (int i = 1; i < 20; i++)
        {
            System.out.println("Retrying lock, loop "+i);
            if (em.tryLock(company, lockObject,
                JadeConstants.LockType.EXCLUSIVE,
                JadeConstants.LockDuration.SESSION, 5))
            {
                System.out.println(
                    "retry was OK, item is now locked");
                return JadeConstants.ExceptionAction.CONTINUE;
            }
            // sleep one second and then try again
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
        System.out.println("retry failed, update aborted");
        return JadeConstants.ExceptionAction.ABORT;
    }
}

/** Creates a new instance of Main */
public Main()
{
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args)
{
    Main main = new Main();
    main.init();
    // add this as a lock exception handler
    main.em.addLockExceptionHandler(main);
    try {
        main.update();
    } catch (RuntimeException ex) {
        ex.printStackTrace();
    } finally {
        // need to remove lock exception handlers before
        // closing
        main.em.removeLockExceptionHandler(main);
        main.closeDown();
    }
}
}

```

3. Build this Java project.
4. Perform the following actions.
  - a. Ensure that the **path** and **java.library.path** variables are set as in step 7 of Appendix A.
  - b. With the JADE database server running (if running the JADE environment in multiuser mode), run the Java project.

If running the JADE environment in single user mode, make sure that the JADE environment is not already running.

This should output the following on successive runs as the address is updated to these two values.

```
Client Roger Boeing 167 Wingtip Road
Client Roger Boeing 999 Wingtip Road
```

5. To see the lock exception handler in action, run a JADE development environment in multiuser mode and then add the following **JadeScript** method to the JADE **ErewhonInvestmentsModelSchema** schema.

```
vars
  s : String;
  company : Company;
  client : Client;
begin
  company := Company.firstInstance;
  client := company.allClients["Roger Boeing"];
  exclusiveLock(client);
  s := "wait";
  read s;
end;
```

6. Run this **JadeScript** method and leave it sitting waiting for user input.

This holds an exclusive lock on the client object for **Roger Boeing**, which the Java code also locks.
7. If you have been running the JADE environment in single user mode before now, change the **persistence.xml jade.database.multiuser** property value to **true**.

Run the Java application again and wait for the lock timeout, which defaults to ten seconds. After this time has elapsed, the console messages from the **exceptionRaised** method (**Retrying lock, loop 1**, and so on) should be displayed.
8. To unlock the client object, enter anything in the JADE user input.

The Java console should now display the following.

```
retry was OK, item is now locked
```

## Further Examples

The **Erewhon** Java shop example in the `examples\erewhon\JavaShopExample` directory provides an example in the `forms.FormShopCheckout` class `doUpdateViaMethod()` method of using transient objects as parameters to a JADE method. This method uses **OrderProxy** instances, which are regular Java objects.

Each **OrderProxy** instance has a reference to a Java **SaleItem** object that contains the details of each retail and tender sale transaction and which has an equivalent JADE transient object. The `doUpdateViaMethod()` method passes these **SaleItem** Java objects to a JADE method and their persistence is done in JADE code using the equivalent JADE objects, rather than using the Java API directly.

Two JADE **Application** class methods are called, as follows.

- The first method, `addToShoppingCard`, passes each **SaleItem** instance to JADE, to add to the internal shopping cart list
- The second method, `processShoppingCart`, processes these objects in JADE to persist them

This processing relies on class `forms.FormLogon` calling the **Application** class `initShopJava` method, passing the selected client object as a parameter. This method also does the equivalent processing of the JADE **Application** class `initialize` method in a standard JADE system.

When a JADE application is started from an external source such as the Java Persistence API, any declared application `initialize` method is not executed automatically, so it must be executed from your code.

---

# Appendix C

# Differences between JADE and Java

---

## Semantic Differences

This appendix describes the semantic differences between JADE applications and Java applications.

### Query

All access in JADE is via references; there is no query language as there is in a relational database.

Collections and references become much more important, as access to objects is done by following references from parent objects. Queries can still be done by being selective in the navigation based on the properties of the objects found, to get the equivalent of an **SQL where** clause.

### Relationships

One-to-many to many-to-one relationships work in an equivalent manner to the Java Persistence API (JPA) but with additional functionality to set parent/child relationships and manual/automatic updating.

### Inverses

The JADE object manager can automatically maintain inverses in relationships between two objects, in a similar way to that of the JPA.

Although the JADE *Java Developer's Reference* covers this in more detail, the following subsections illustrate the differences with a combination of relational and one-directional and bi-directional options.

#### Relationship One-to-One, Uni-directional

The **Customer** class has exactly one **Address**, and **Address** has exactly one **Customer** but with no reference.

For both Java and JADE, the relationship is created by the following.

```
Customer cust.setAddress(address);
```

## Java Handling

```
@Entity
public class Customer {

    @OneToOne
    @JoinColumn(name="ADDRESS_ID")
    public Address getAddress()
        etc
}

@Entity
public class Address {
    street, city etc with no specific annotation and no Customer reference
}
```

## JADE Handling

```
@Entity
public class Customer {
    @OneToOne()
    public Address getAddress()
        etc
}

@Entity
public class Address{
    street, city etc with no specific annotation and no Customer reference
}
```

## Relationship One-to-One, Bi-directional

The **Customer** class has zero or one **CreditCard**, and **CreditCard** has a reference back to **Customer**.

## Java Handling

```
@Entity
public class CreditCard {
    @OneToOne(mappedBy="creditCard")
    public Customer getCustomer()
        etc
}

@Entity
public class Customer {
    @OneToOne
    @JoinColumn(name="CREDIT_CARD_ID")
    public CreditCard getCreditCard()
        etc
}
```

The **@OneToOne(mappedBy)** annotation on **customer** in **CreditCard** marks this as the *inverse* property of the relationship and **Customer** as the *owning* side of the relationship.

The values of **both** sides of the relationship need to be set in memory for the relationship to be updated; that is, there is no automatic inverse maintenance.

## JADE Handling

```

@Entity
public class CreditCard {
    @OneToOne(relationshipType=ReferenceRelationshipType.CHILD,
        updateMode=ReferenceUpdateMode.AUTOMATIC,inverse="creditCard")
    public Customer getCustomer()
        etc
}

@Entity
public class Customer {
    @OneToOne(relationshipType=ReferenceRelationshipType.PARENT,
        updateMode=ReferenceUpdateMode.MANUAL,inverse="customer")
    public CreditCard getCreditCard()
        etc
}

```

Each **@OneToOne()** annotation has an explicit **relationshipType** attribute setting **Customer** as the owning, or parent, side of the relationship, and an explicit **inverse** property and **updateMode** attribute for both sides of the relationship.

Because **Customer** has an **updateMode** of manual, setting the **Customer creditCard** value automatically sets the **CreditCard customer** value to the equivalent **Customer** reference.

An alternative definition of the relationship in JADE is as follows.

```

@Entity
public class CreditCard {
    @OneToOne(relationshipType=ReferenceRelationshipType.PEER,
        updateMode=ReferenceUpdateMode.MAN_AUTO,inverse="myCreditCard")
    public Customer getCustomer()
        etc
}

@Entity
public class Customer {
    @OneToOne(relationshipType=ReferenceRelationshipType.PEER,
        updateMode=ReferenceUpdateMode.MAN_AUTO,inverse="myCustomer")
    public CreditCard getCreditCard()
        etc
}

```

This changes the relationship of the two properties so that neither is the owning side of the relationship (that is, it is a *peer* relationship), but still only one side of the relationship needs to be set.

Setting either property (the manual side of the **MAN\_AUTO** update mode) in code causes the JADE object manager to automatically (the automatic side) set the other property value to match.

## Relationship One-to-Many, Uni-directional

The **Customer** class has zero to many **Phones**.

## Java Handling

```
@Entity
public class Customer {
    @OneToMany()
    @JoinColumn(name="CUSTOMER_ID")
    public Collection<Phone> getPhones()
        etc
}

@Entity
public class Phone {
    number, type etc with no reference to Customer
}
```

The relationship between **Customer** and **Phone** must be explicitly set; for example, to add a phone, as follows.

```
Phone phone = new Phone();
cust.getPhones().add(phone);
```

The following example removes a phone.

```
cust.getPhones().remove(phone);
entityManager.remove(phone);
```

## JADE Handling

```
@Entity
public class Customer {
    @OneToMany()
    public PhoneArray<Phone> getPhones()
        etc
}

@Entity
public class Phone {
    number, type etc with no reference to Customer
}
```

The same applies in JADE; that is, the relationship between **Customer** and **Phone** must be explicitly set; for example, to add a phone, as follows.

```
Phone phone = new Phone();
cust.getPhones().add(phone);
```

The following example removes a phone.

```
cust.getPhones().remove(phone);
entityManager.remove(phone);
```

## Relationship One-to-Many, Bi-directional

The **Customer** class has zero to many **Phones**. **Phone** has a reference back to **Customer**.

## Java Handling

```
@Entity
public class Customer {
    @OneToMany(mappedBy="owner")
    public Collection<Phone> getPhones
        etc
}

@Entity
public class Phone {
    number, type etc
    @ManyToOne()
    @JoinColumn(name="CUSTOMER_ID")
    public Customer getOwner
        etc
}
```

The many-to-one side of the relationship is the owner and the relationship between **Customer** and **Phone** must still be explicitly set on both sides; for example, to add a phone, as follows.

```
Phone phone = new Phone();
cust.getPhones().add(phone);
phone.setCustomer(cust);
```

The following example removes a phone.

```
entityManager.remove(phone);
```

## JADE Handling

```
@Entity
public class Customer {
    @OneToMany(relationshipType=ReferenceRelationshipType.PARENT,
        updateMode=ReferenceUpdateMode.AUTOMATIC,inverse="owner")
    public PhoneArray<Phone> getPhones()
        etc
}

@Entity
public class Phone {
    number, type
    @ManyToOne(relationshipType=ReferenceRelationshipType.CHILD,
        updateMode=ReferenceUpdateMode.MANUAL,inverses={"allPhones"})
    public Customer getOwner() }
    etc
}
```

In JADE, the relationship between **Customer** and **Phone** must be set from the child side only by setting the inverse owner property on **Phone**; for example, to add a phone, as follows.

```
Phone phone = new Phone();
phone.setOwner(cust);
```

This automatically adds the phone to the **cust.phones** collections. To remove one, only the **phone** deletion is required, as follows. (It is automatically removed from the **cust.phones** collection.)

```
entityManager.remove(phone);
```

## Fetch Depth and Lazy Instantiation

In the JPA and in some other object databases, a significant problem is that of determining how deeply to follow references as each object is instantiated from the (relational or object) database. If the primitive values only are read, each reference must be separately read when it is first referenced. This requires that:

- The reference is implemented as a proxy of the actual reference, with extra state to indicate if it is instantiated or not.
- Extra Java byte code is added to each access to check for instantiation and do it as required.

Whichever action is taken, the instantiation requires an extra trip to the database. This can lead to the **n+1** performance problem. When reading an object (that is, the **1** read), all instances of a collection reference from that object (that is, the additional **n** collection instances) is done using **n+1** database calls, which can be much slower than reading all instances together in the initial object read. However, if the object has many references and collections, and the objects referenced also have references, a lot of data is read and many objects are instantiated when only some may actually be used.

The JPA has two ways of handling this, as follows.

- On the JPA annotation for each reference definition, **FetchType** with values of **EAGER** or **LAZY**.  
This defaults to **EAGER** for one-to-one relationships and to **LAZY** for one-to-many relationships. It is, however, only a hint to the JPA provider.
- Using the query API, where the **LEFT JOIN FETCH** option is added to a **SELECT**, to load a collection along with the base object.  
This, however, requires an explicit query and it at least partially negates the benefit or point of defining the collection or reference in the first place.

JADE has no need for a fetch depth, as all JADE references are implicitly lazy. When an object is fetched by the JADE object manager from the database, all reference properties of the object are present internally as **oid** values. The oid is not resolved to the equivalent object until the first time this reference is used.

The JADE object manager then checks the object cache for the instance and if it is not found, the object is requested from the database server and placed into the local object cache. The object manager does not return with the referenced object until this has completed internally, so that it acts as if the object were already present.

If the reference is to a collection, collections are composite objects made up of collection blocks, which are lists of oid values for the collection contents. Each block holds a configurable number of entries, so that iterating through a collection brings the entry references this number at a time. Each collection entry is read from the database only when the object is first accessed, so the objects in each collection block are fetched from the database only as each is accessed. This means that collections containing millions of entries can be efficiently read, as there is built-in paging of the contents.

All of this processing is done automatically by the JADE object manager and is automatically available to the equivalent Java objects.

## Cascading

In the Java Persistence API, reference properties can have an **@Cascade** annotation applied. This means that when the code applies an entity manager operation on an entity bean instance, it automatically performs the same operation on this relationship property.

In JADE, the nearest equivalent is the automatic processing of inverses in relationships. When the manual side of a relationship is changed, the JADE object manager also automatically actions the automatic side; for example, setting an inverse reference can also populate a collection. Another example is that deleting an object that is the parent of a relationship also automatically deletes the children objects. This processing is done automatically by the JADE object manager.

## Object Lifetime

Because JADE is oriented around persistence, all objects have a lifetime. For a persistent object, this is until it is explicitly deleted, like an RDMS row. For a transient object, this is while the owning process is running. A transient object that is no longer required can also be deleted, using the **EntityManager** class **remove** method, as for a persistent delete. This differs from Java objects, which are automatically garbage-collected some time after they go out of scope.

In JADE, unused transient objects are also effectively garbage-collected out of the object cache memory, but to a temporary file in case they are required later.

JADE transient objects would normally be created from Java objects for use as parameters to existing JADE methods, where the existing JADE code does the required processing and persisting of the Java data. These objects are created using the following **EntityManager** class method.

```
createObject(object, JadeConstants.ObjectLifetime.TRANSIENT);
```

If a Java object is therefore created with a corresponding JADE transient object, when the object is no longer used it should have an **EntityManager** class **remove(obj)** method called on it to delete the JADE object.

## Object Deletes

When a persisted Java object has **em.remove(obj)** called on it, the corresponding JADE object is deleted. If the JADE object is persistent, this remove call must be done in transaction state and the JADE object is removed from the database when the subsequent **EntityTransaction** class **commit()** method call is done. If the JADE object is transient, it is deleted from the local JADE cache.

In either case, the field of the Java object with the **@Id** annotation is reset to zero (**0**), to indicate that the corresponding JADE object no longer exists.

## Object Identity

JADE objects have a unique identity using their **oid** property.

Java objects managed by JADE are created and read so that each unique JADE object has a single equivalent Java object. This uniqueness is for each **EntityManager** instance.

JADE-managed Java object equality should then be compared using the **oid** values; that is, the **equals()** method should check for equality of the **oid** value. Objects inheriting from **com.jadeworld.jade.rootschema.Object** have such an **equals** method.

## Detached Objects

The JADE Persistence API does not automatically detach objects when the owning **EntityManager** is closed, as the JPA does.

Each Java object holds the oid of the corresponding JADE object in the field with the **@Id** annotation and this is the main thing needed to relate the Java and JADE objects. Java classes with **DbField** fields can be used independently of JADE, as long as no **DbProperty** fields are also referenced, because the **DbField** items are just regular Java fields. These objects can be used as Data Objects and transferred between systems, to meet your requirements. The **EntityManager** class **refresh** and **persist** methods can be used to synchronize the Java values with the JADE ones in either direction, if required, without needing an explicit detached or attached status.

## Exceptions

Exceptions from JADE are trapped in the JNI interface and re-thrown as **com.jadeworld.jade.JadeException** instances. JADE exceptions have a unique number, which is returned by the **getErrorCode()** method. This numeric value is used in code to determine what action to take from the exception.

For details of each error type, see the *JADE Error Messages and System Messages* document.

## Lock Exceptions

Lock exceptions are raised from the JADE object manager when an attempt is made to lock an object that is already locked by another process. These exceptions are potentially recoverable in JADE.

To achieve this in the Java code, a user class needs to implement the **com.jadeworld.jade.LockExceptionHandler** interface, which has one method: **exceptionRaised(JadeException exceptionDetails)**. This method returns a **JadeConstants.ExceptionAction** enum. To register this class as a lock exception handler, use the **EntityManager.addLockExceptionHandler(obj)** method.

To deregister this class, use the **removeLockExceptionHandler(obj)** method. The handlers must be deregistered before an **EntityManager** instance can be closed.

When a lock exception is raised, the **exceptionRaised** method is called for each registered **LockExceptionHandler** instance, from the most-recently registered up to the first registered instance. The **exceptionDetails** parameter contains the details of the lock exception.

Each handler can deal with the exception and return **ExceptionAction.ABORT** or **ExceptionAction.CONTINUE**, or it can return **ExceptionAction.PASS\_BACK**, to let the next handler in turn deal with it.

If none of the handlers in the registered stack handle it (that is, there are no registered handlers or they all return **ExceptionAction.PASS\_BACK**), it is raised as a regular **JadeException**.

For an example of a lock exceptions handler, see the **Main** class in the example in Appendix B of this document.

## Run Time Handling

This section covers object handling at run time.

### Transaction State and Persisting

Because JADE objects are persisted from the in-memory cache copy, your application must enter transaction state before making any change to a persistent object, because the object needs to be locked to prevent simultaneous changes. This differs from the Java Persistence API (JPA), where changes can be made outside of transaction state and queued up to be sent to the database from a subsequent transaction.

All changes made to **DbProperty** fields are applied to the corresponding JADE object when the property is changed, so these are automatically persisted when a transaction commit is done.

All changes made to **DbField** fields are applied only to the corresponding JADE object when an **EntityManager** class **persist()** method of the object is called, before a transaction commit. This allows the object to be used as a regular Java object (POJO) and changes made to suit the local processing without these changes being automatically persisted. The manual persist call can then be done for those objects that are required to be persisted.

A JADE commit updates the database to reflect all changes made to persistent objects within the transaction. If automatic cache coherency is enabled, it also sends internal notifications to other nodes for them to update their cached copies of these objects on the next object reference.

### Rollback

A Java Persistence API rollback invalidates the current entity manager cache so that the entity manager needs to be cleared or a new instance created, and the existing objects re-managed before they can be processed again in any retry code. This can be difficult in a complex transaction.

In JADE, a deleted object is reinstated in the database following a rollback, but the reference used in the delete command is still null, as the delete command (that is, the **EntityManager** class **remove()** method) sets the reference to null.

A changed object reverts back to its prior state in the database and the cached copy is marked as obsolete, so that any access of a property results in the object being automatically refreshed from the database before the property value is returned.

A newly created instance reference that is then rolled back is non-null, and is an invalid reference. Attempting to de-reference the instance therefore raises a **JadeException** instance with an **errorCode** property value of **4**.

A second reference to a deleted item is the same as a rolled-back changed item, as it now reflects the restored value from the database.

### Locking

JADE locking is integrated between the database and the cache copies. The lock manager is centralized at the database server. For details, see Chapter 9 of the *JADE Developer's Reference*.

Briefly, there are different levels of locking that can be applied to an object: **shared**, **reserve**, and **exclusive**. You can manually apply locks by using methods in the **EntityManager** and **rootschema.Object** classes. Locks are also automatically applied.

You can manually unlock objects by using the **unlock** method.

Locks are automatically applied by the JADE object manager for different actions; for example, updating a persistent object.

You can use the **EntityManager** class **beginLoad** and **endLoad** method pair and the **beginLock** and **endLock** method pair to bracket a block of access logic and defer unlocking until the **endLoad** or **endLock** method is called. For details, see Chapter 1 of the JADE *Developer's Reference*.

## Optimistic Locking

The JPA implements optimistic locking, by defining a numeric or timestamp property for a class and using the **@Version** annotation. The **EntityManager** class **lock** method for a read lock saves the current value of this field for the specified object. For a write lock, it increments or updates it to the database and saves the value. When a subsequent update of this object is done, the SQL update is built as **update xxx where primary-key = pk and version=saved-value and other changes including incrementing the version**.

The persistence provider then checks the returned number of rows changed. If the version field has not been changed by any other process, the value is **1**, indicating that the update has been successful.

If the number is zero (**0**), the version has been subsequently changed by some other process, and the persistence provider raises a Stale Data exception. At this point, the update processing must be re-done using a new **EntityManager** instance (for details, see "Recovery from Persistent Provider Exceptions", later in this document).

This allows logical locking of an object across transactions; for example, an initial fetch to show and use it on a number of input forms before the eventual update, without having to physically lock the database row.

The JPA does not have a mechanism for a pessimistic lock. Pessimistic locks can be done by explicit SQL queries or by non-standard query hint settings such as those in TopLink.

The equivalent can be done with the underlying JADE object by saving the **edition** property (from the **EntityManager.edition()** method) when the object is first accessed. This property is automatically updated by the JADE object manager when a JADE object is updated so the update code can lock the object for update and check that the edition is still the same. If not, the logic can take whatever action is required, without having to re-do all of the processing.

## Caching

JADE has three internal cache managers: one for persistent objects, one for transient objects, and one for remote transient objects. The transient cache is divided up by thread.

For regular objects, the object is added to the cache when first read. The cache is updated from the server on receipt of a system notification for this object or when the object is locked.

Collection headers and blocks have a **share** lock applied by the JADE object manager when they are read, to ensure that they are always current.

JADE has a cache coherency option, which is enabled by default so that when an object is updated in another JADE node, including persisted Java objects, it is automatically reloaded in the cache of the current node.

## Notifications

Notifications are used extensively in JADE systems to communicate changes of state between different parts of a JADE system.

Notifications are used to inform users about events and to force the updating of cache. When a system notification is received, the buffer is marked as obsolete if the object that caused the notification (the target object) is in cache. If the user then tries to access the object, the JADE object manager checks the edition with the server and if it is not current, requests the latest version to be sent from the server.

Notifications are of two types, as follows.

- System notification events, which are caused when target objects (or classes) are created, updated, or deleted, and are sent automatically by the JADE database server following the commit.
- User notification events, which are caused by user code calling the **EntityManager** class **causeEvent** method.

Notifications must be registered for before they can be received, by:

- An entity class implementing the **com.jadeworld.jade.SystemNotification** interface with a single method **systemNotification** to receive system notifications
- Implementing the **com.jadeworld.jade.UserNotification** interface with a single **userNotification** method to receive user notifications

## Database Connections

Java persistence to relational databases generally uses a pool of JDBC connections for performance.

As each JPA **EntityManager** instance can use a new connection, the connection pool can get used up if many **EntityManager** instances are in use at once. Because the use of these connections is generally seen as somewhat expensive, as much as possible is done each time one is used, by batching up inserts and updates, and issuing SQL selects that read all the required data in one request, if possible.

In JADE, the internal connection from a JADE client node to the JADE database server is light-weight, and there is less cost than an equivalent JDBC connection. There is no explicit equivalent to a pool of connections, as JADE handles this internally.

The JADE database server node has a pool of threads for servicing requests from client nodes, which you can configure using JADE initialization file parameters. (For details, see the JADE *Initialization File Reference*.) However, there is still an advantage in minimizing the number of trips to the server in JADE, particularly as lock and unlock requests also involve a message to the server and back.

## Recovery from Persistence Provider Exceptions

When an exception has been raised from persistence provider products such as Hibernate and TopLink, no further processing should be done with the current **EntityManager** instance. Objects managed by this instance are not guaranteed to be in a usable state after any such exception.

Your code must create a new **EntityManager** instance and repeat the processing.

There is no such requirement in JADE. The corresponding JADE objects remain managed and usable, except that the object that caused the exception may need different processing, depending on the type of exception.