6 Legacy Connectivity

CERTIFICATION OBJECTIVES

6.01	Distinguish Appropriate from Inappropriate Techniques for Providing Access to a Legacy System from Java Technology Code Given an Outline Description of That Legacy System	✓	Two-Minute Drill
		Q&A	Self Test

The capacity and capability to migrate legacy systems to Java 2 Enterprise Edition (JEE) is on the increase as the need to web-enable legacy systems increases. A growing number of legacy systems, including IBM mainframe, UNIX, and client/server, can now be migrated to or integrated with JEE to take advantage of its security, speed, reliability, and cross-platform capabilities. Some of the benefits of this are freedom from obsolete software, return on the original investment in legacy systems (especially after Y2K) via extended life of these systems, and opportunities for e-commerce using legacy systems and databases. To that end, this chapter will cover the following topics:

- Engineering the Enterprise Information Systems (EIS) Integration Tier
- Best practices for EIS integration
- Guidelines for data access
- EIS access objects and connections
- Java Enterprise Engineering: Services
- Role of transactions
- Best practices relating to transactions in each tier
- Appropriate and inappropriate use for given situations

Introduction to Legacy Connectivity

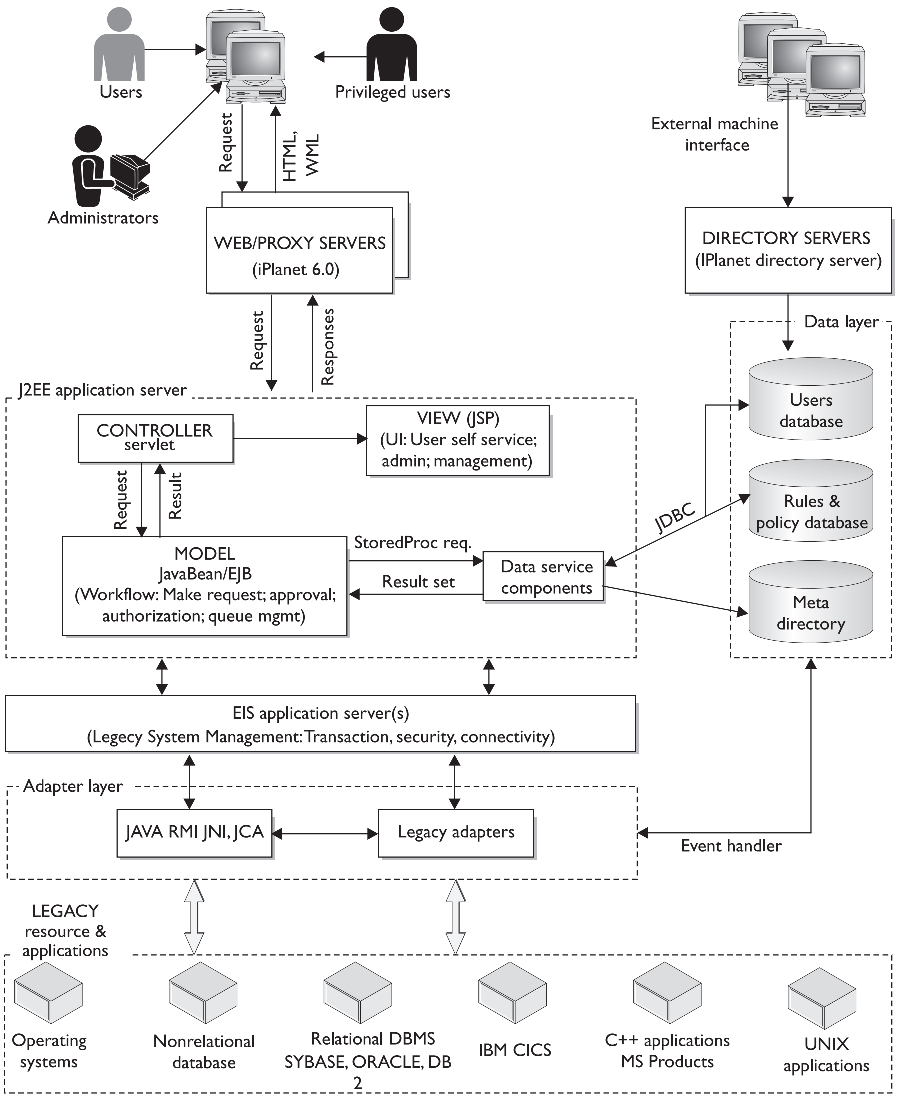
As businesses move toward an e-business strategy, the challenge of legacy connectivity is to enable each enterprise to integrate new e-business applications with existing enterprise information systems (EISs). Enterprise applications require access to applications running on an EIS. These systems provide the information infrastructure for an enterprise—the so-called "books and records," as they say on Wall Street.

EISs include enterprise mainframe transaction processing systems, relational database management systems (RDBMS), and other legacy information systems. Enterprises run their businesses using the information stored in these systems, and the success of an enterprise critically depends on this information. Enterprises with successful e-businesses need to integrate their EISs with web-based applications. They need to extend the reach of their EISs to support business-to-business (B2B) transactions.

Before the JEE Connector Architecture (JCA) was defined, no specification for the Java platform addressed the problem of providing a standard architecture for integrating an EIS. We used JNI (Java Native Interface) and RMI (Remote Method Invocation) to create a Java interface to a process running in its native domain. For example, a Java program using JNI, RMI, or CORBA (Common Object Request Broker) can call a C++ program running on a Windows NT machine. Most EIS vendors as well as application server vendors use nonstandard proprietary architectures to provide connectivity between application servers and enterprise information systems that provide services such as messaging, legacy database access, and mainframe transaction or batch processing. Figure 6-1 illustrates the complexity of an EIS environment.

FIGURE 6-1

EIS environment: legacy applications with an e-business front end


Users
Privileged users
Administrators
Request
HTML, WML
External machine interface
WEB/PROXY SERVERS (iPlanet 6.0)
DIRECTORY SERVERS (IPlanet directory server)
Request
Responses
Data layer
J2EE application server
CONTROLLER servlet
VIEW (JSP) (UI: User self service; admin; management)
Users database
Request
Result
Rules & policy database
JDBC
MODEL JavaBean/EJB (Workflow: Make request; approval; authorization; queue mgmt)
StoredProc req.
Result set
Data service components
Meta directory
EIS application server(s) (Legecy System Management: Transaction, security, connectivity)
Adapter layer
JAVA RMI JNI, JCA
Legacy adapters
Event handler
LEGACY resource & applications
Operating systems
Nonrelational database
Relational DBMS SYBASE, ORACLE, DB 2
IBM CICS
C++ applications MS Products
UNIX applications

Legacy Connectivity Using Java: the Classic Approach

Thus far, the classic approach to legacy connectivity is based on the two-tier client/server model, which is typical of applications that are not based on the web. With this approach, an EIS provides an adapter that defines an application programming interface (API) for accessing the data and functions of the EIS—basically, you "black-box" the target system and create a Java API. A typical client application accesses data and functions exposed by an EIS through this adapter interface. The client uses the programmatic API exposed by the adapter to connect to and access the EIS. The adapter implements the support for communication with the EIS and provides access to EIS data and functions.

Communication between an adapter and the EIS typically uses a protocol specific to the EIS. This protocol provides support for security and transactions, along with support for content propagation from an application to the EIS. Most adapters expose an API to the client that abstracts out the details of the underlying protocol and the distribution mechanism between the EIS and the adapter. In most cases, a resource adapter is specific to a particular EIS. However, an EIS may provide more than one adapter that a client can use to access the EIS. Because the key to EIS adapters is their reusability, independent software vendors try to develop adapters that employ a widely used programming language to expose a client programming model that has the greatest degree of reusability.

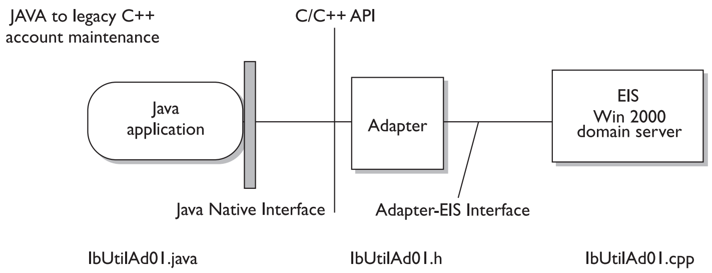
Using a Simple EIS Java Adapter

An EIS may provide a simple form of an adapter, where the adapter maps an API that is specific to the EIS to a reusable, standard API. Often, such an adapter is developed as a *library*, whereby the application developer can use the same programming language to access the adapter as she uses to write the application, and no modifications are required to the EIS. For example, a Java application developer can use a Java-based adapter—an adapter written in the Java programming language—to access an EIS that is based on some non-Java language or platform.

An EIS adapter may be developed as a C library. For example, the code in Figure 6-2 illustrates a Java application that uses a JNI to access this C library or C-based resource adapter. The JNI is the native programming interface for Java, and it is part of the Java Developers Kit (JDK). The JNI allows Java code that runs within a Java Virtual Machine (JVM) to operate with applications and libraries written in other languages, such as C and C++. Programmers typically use the JNI to write native methods when they cannot write the entire application in Java. This is the case when a Java application needs to access an existing library or application written in another

FIGURE 6-2

Java JNI application



programming language. While the JNI was especially useful before the advent of the JEE platform, some of its uses may now be replaced by the JEE Connector Architecture. As you can see in Figure 6-2, the JNI to the resource adapter enables the Java application to communicate with the adapter's C library. While this approach does work, it is complex to use. The Java application has to understand how to invoke methods through the JNI. This approach also provides none of the JEE support for transactions, security, and scalability. The developer is exposed to the complexity of managing these system-level services, and must do so through the complex JNI.

```
public class lbUtilAd01  {
public native String createUser  (String pszUIDName, String pszUIDPassword,
String pszFirstName,String pszLastName,
 String pszOrg,String pszRoot,
 String pszAdminName, String pszAdminPassword);
  static
  {
      // Load the C++ DLL
      System.loadLibrary("lbUtilAD01");
  }
  public static void main(String args[])
  {
    lbUtilAd01 AD01 = new lbUtilAd01();
  AD01.createUser("Jbambara", "test1234", "Joe", "Bambara", "ou=Test OU",

"dc=TRADING, dc=bank, dc=com", "administrator", "pw1234$!");
Output of JAVAH compiler: javah lbUtilad01
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class tacadapter_lbUtilAd01 */
#ifndef _Included_tacadapter_lbUtilAd01
#define _Included_tacadapter_lbUtilAd01
#ifdef __cplusplus
```

```
extern "C" {
#endif
JNIEXPORT jstring JNICALL Java_tacadapter_lbUtilAd01_createUser
  (JNIEnv *, jobject, jstring, jstring, jstring, jstring, jstring, jstring,

jstring,
string);
```

Here's the C++ program *lbUtilAD01.cpp*, which is called by *lbUtilAD01.java*:

```
// lbUtilAD01.cpp: implementation of the lbUtilAD01 class.
// This will CREATE user WINNT account for MS ADSI
….
#include "tacadapter_lbUtilAd01.h"
#define _WIN32_WINNT 0x0500
extern "C" __declspec( dllexport ) LPWSTR CharStringToUnicodeString
(const char *string);
char *GetSID(const char *szDomainName,LPWSTR,LPWSTR,const
char *szUserName,VARIANT *);
// JAVA JNI interface call signature
JNIEXPORT jstring JNICALL Java_tacadapter_lbUtilAd01_createUser
 (JNIEnv *env, jobject obj, jstring pszUIDName, jstring pszUIDPassword,jstring
szFirstName,jstring pszLastName,jstring pszOrg,jstring pszRoot,jstring

pszAdminName, jstring pszAdminPassword)
{
 char strORG[1024],strRDN[1024], strFullName[1024];
 HRESULT result;
 jstring rMessage;
 // convert call signature args to use in program
 const char *szUIDName       = env -> GetStringUTFChars(pszUIDName, 0);
 const char *szUIDPassword   = env -> GetStringUTFChars(pszUIDPassword, 0);
 const char *szOrg   = env -> GetStringUTFChars(pszOrg, 0);
 const char *szRoot  = env -> GetStringUTFChars(pszRoot, 0);
 const char *szAdminName     = env -> GetStringUTFChars(pszAdminName, 0);
 const char *szAdminPassword = env -> GetStringUTFChars(pszAdminPassword, 0);
 const char *szFirstName     = env -> GetStringUTFChars(pszFirstName, 0);
 const char *szLastName      = env -> GetStringUTFChars(pszLastName, 0);
 IADsContainer *pContainer;
 IADs *pServer=NULL;
 IADsUser *pADuserpw=NULL;
 IDispatch *pDisp=NULL;
LPWSTR lpADSIPath,lpUIDName,lpUIDPasswd,lpFirstName,lpLastName;
LPWSTR lpOrg,lpFullName, lpRDNName;
// GET ADSIPATH
lpADSIPath = CharStringToUnicodeString(strORG);
result = ADsOpenObject(lpADSIPath,lpAdminID,lpAdminPasswd,
ADS_SECURE_AUTHENTICATION, IID_IADsContainer,(void**)&pContainer);
// CREATE USER
lpRDNName = CharStringToUnicodeString(strRDN);
 result= pContainer->Create(L"user",lpRDNName,&pDisp);
 result = pADuserpw->SetPassword(lpUIDPasswd);
//  COMMIT the changes
 result=pADuserpw->SetInfo();
    if (!SUCCEEDED(result))
    {
  rMessage = env -> NewStringUTF(GetErrorCode(result));
  cout << "Fail to pw set info" << endl;
  return(rMessage);
    }
```

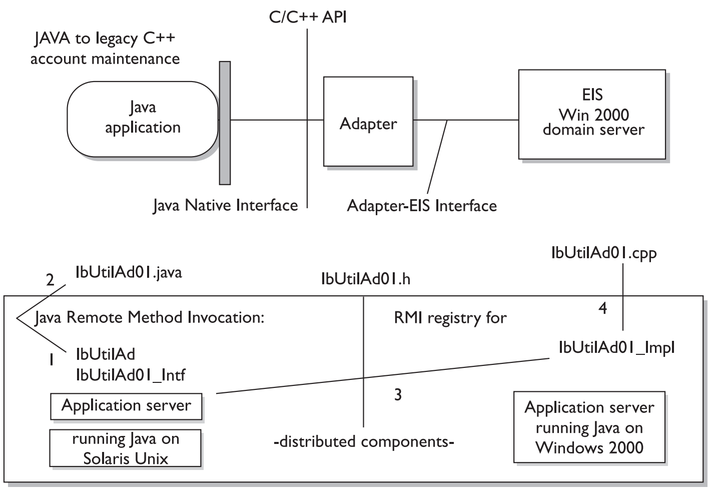
Distributed EIS Adapters

Another, more complex, form of an EIS adapter might do its "adaptation" work across diverse component models, distributed computing platforms, and architectures. For example, an EIS may develop a distributed adapter that includes the ability to perform remote communication with the EIS using Java RMI. This type of adapter exposes a client programming model based on component-model architecture. Adapters use different levels of abstraction and expose different APIs based on those abstractions, depending on the type of the EIS. For example, with certain types of EISs, an adapter may expose a remote function call API to the client application. If so, a client application uses this remote function call API to execute its interactions with the EIS. An adapter can expose either a synchronous or an asynchronous mode of communication between the client applications and the EIS.

In the following code, the *lbUtilAd01* C++ program illustrates the use of adapters designed for synchronous communication using Java RMI. The rmic compiler generates stub and skeleton class files (JRMP protocol) and stub and tie class files (IIOP protocol) for remote objects. These classes files are generated from the compiled Java programming language classes that contain remote object implementations. A remote object is one that implements the interface `java.rmi.Remote`. The classes named in the rmic command must be classes that have been compiled successfully with the javac command and must be fully package qualified. Adapters designed for this approach provide a synchronous request-reply communication model for use between an application and an EIS. In the following example and Figure 6-3, when an application wants to interact with the EIS to create an Windows 2000 account, it invokes this remote function on the EIS. The application that initiated the call and then waits until the function completes

FIGURE 6-3

Java RMI application



and returns its reply to the caller. The reply contains the results of the function's execution on the EIS. An interaction such as this is considered *synchronous* because the execution of the calling application waits synchronously during the time the function executes on the EIS. One form of synchronous adapter allows bidirectional synchronous communication between an application and an EIS. This type of adapter enables an EIS to call an application synchronously.

```
import java.rmi.*;
import java.rmi.Naming;
import java.rmi.RemoteException;
public class lbUtilAd {
  String Win2KServer;
  lbUtilAd01Intf obj;
  public lbUtilAd(String server) throws Exception {
        Win2KServer = server;
     obj = (lbUtilAd01Intf)Naming.lookup("//"+Win2KServer+"/lbUtilAd01");
  }
public String createUser

(String pszUIDName, String pszUIDPassword,

String pszFirstName,String pszLastName,

String pszOrg,String pszRoot,

String pszAdminName, String pszAdminPassword)
throws RemoteException
  {
       return (obj.createUser(pszUIDName, pszUIDPassword,

pszFirstName, pszLastName, pszOrg, pszRoot,  pszAdminName, pszAdminPassword));
  }
public static void main(String args[]) throws Exception
{
lbUtilAd obj = new lbUtilAd("ADSISERVER");
obj.createUser("jbambara", "pw", "Joe","Bambara", "org1","root1","administrator",
"admpw12$!"));
}
}
```

Legacy Connectivity Using JEE Connector Architecture

The evolving Java Connector Architecture (JCA) standard will obviate most of the need to build JNI and RMI code by providing a mechanism to store and retrieve enterprise data in JEE. The latest versions of many application servers, including BEA WebLogic and IBM WebSphere, support JCA adapters for enterprise connectivity. Using JCA to access an EIS is analogous to using JDBC to access a database. By using the JCA, EIS vendors no longer need to customize their products for each application server. Application server vendors who conform to the JCA need not add custom code whenever they want to obtain connectivity to a new EIS.

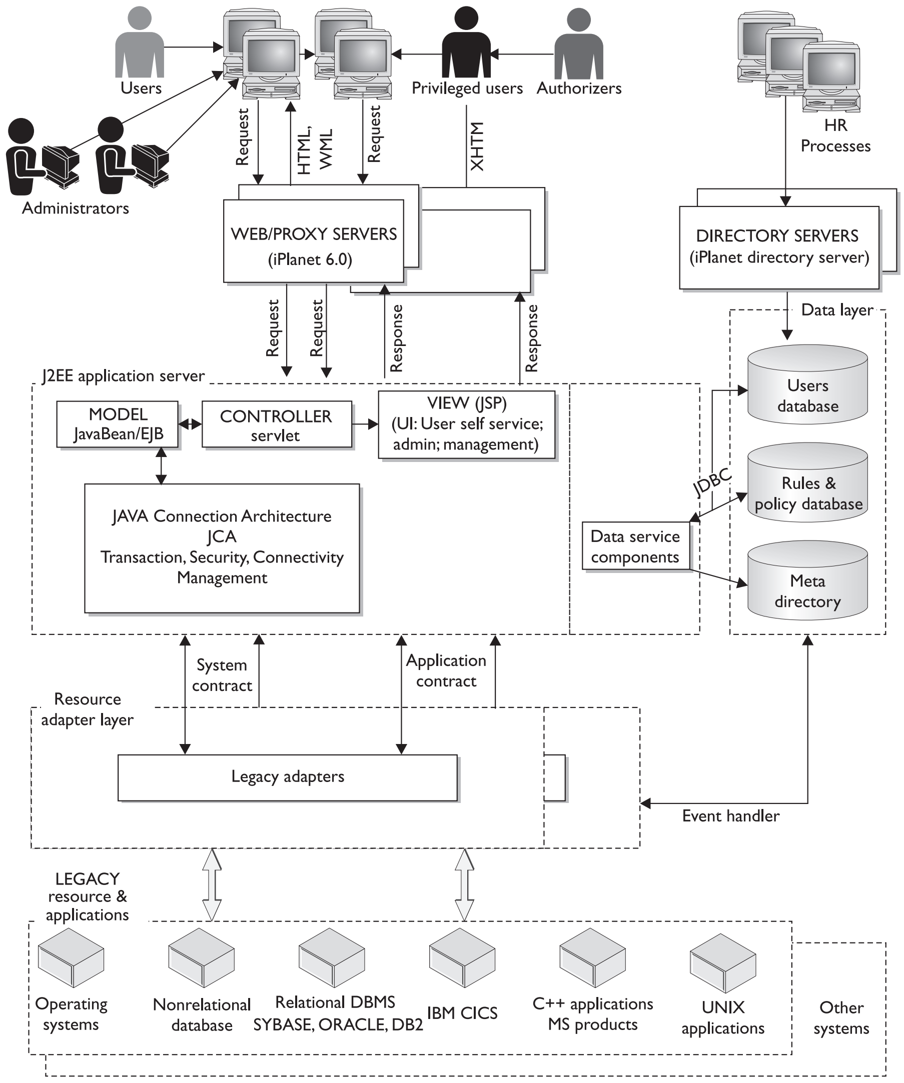
Before JCA, each enterprise application integration (EAI) vendor created a proprietary resource adapter interface for its own EAI product, requiring a resource adapter to be developed for each EAI vendor and EIS combination (for instance, you need a SAP resource adapter to use the messaging tools of Tibco). To solve that problem, as one of its main thrusts, JCA attempts to standardize the resource adapter interfaces. The JCA provides a Java solution to the problem of connectivity between the many application servers and EISs already in existence. The JCA is based on the technologies that are defined and standardized as part of JEE.

The JCA defines a standard architecture for connecting the JEE platform to heterogeneous EISs. Examples of EISs include mainframe transaction processing, such as IBM CICS; database systems, such as IBM DB2; and legacy applications not written in the Java programming language, such as IBM COBOL. By defining a set of scalable, secure, and transactional mechanisms, the JCA enables the integration of EISs with application servers and enterprise applications.

The JCA enables a vendor to provide a standard resource adapter for its EIS. The resource adapter is integrated with the application server, thereby providing connectivity between the EIS and the enterprise application. An EIS vendor provides a standard resource adapter that has the ability to plug into any application server that supports the JCA. Multiple resource adapters, i.e., one per type of EIS, can be added into an application server. This ability enables application components deployed on the application server to access the underlying EISs. Figure 6-4 illustrates the JCA.

FIGURE 6-4

JEE Connector Architecture



Resource Adapter

Deployable JCA components are called resource adapters. Basically, resource adapters manage connections or other resources for interaction with some facility. The definition is open ended, as resource adapters can be used for almost anything. A resource adapter manifests itself as an implementation of interfaces in the *javax.resource.cci* and *javax.resource.spi* packages. It will require a system-level software library when you are accessing a resource that uses native libraries to connect to an EIS. EIS vendors, middleware or application server vendors, or even end users of legacy systems provide a resource adapter. A resource adapter implements the EIS adapter-side of the connector system contracts. In JCA version 1.0, these contracts include connection management, transaction management, and security. In JCA version 1.5, there are additional contracts that we will discuss later in the chapter. A resource adapter also provides a client-level API that applications use to access an EIS. The client-level API can be the common client interface (CCI) or an API specific to the resource adapter or the EIS. A resource adapter can also be used within an application server environment, which is referred to as a *managed* environment. The application server interacts with the resource adapter using the system contracts, while JEE components use the client API to access the EIS. A resource adapter can also be used in a two-tier or nonmanaged scenario. In a nonmanaged scenario, an application directly interacts with the resource adapter, using both the system contracts and the client API to connect to the EIS.

System Contract

A *contract* is an agreement between parties to provide collaborative, mutually beneficial interactions. An application server and an EIS collaborate to keep all system-level mechanisms, such as transactions, security, and connection management, transparent to the application components. As a result, an application component provider can focus on the development of business and presentation logic for its application components and need not resolve at the system level issues related to EIS integration. This facilitates development of scalable, secure, and transactional enterprise applications that require connectivity with multiple EISs.

- The benefits of the system contract are the *quid pro quo* provided by the resource adapter. It is the set of functionality you get to help perform the business task.
- *Connection management contracts* provide for pool connections to an underlying EIS and let application components connect to an EIS. This leads to a scalable application environment that can support a large number

of clients requiring access to EISs. Connection management enables the application server to maintain back-end system connections. Support for connection pooling is provided, since creating connections to back-end systems is expensive. Connection pooling enables an EJB server to pool connections to back-end systems, so rather than opening connections on an as-needed basis, connections with data and services are established, configured, cached, and reused automatically by the application server. The contract enables an application server to offer its own services for transaction and security management.

- A *transaction management contract* between the transaction manager and an EIS that supports transactional access to EIS resource managers lets an application server use a transaction manager to manage transactions across multiple resource managers. The contract also supports transactions that are managed internal to an EIS resource manager without the necessity of involving an external transaction manager. The transaction management contract supports transactional access to underlying resource managers. The service enables the transaction manager provided within the Enterprise JavaBeans (EJB) server to manage transactions across multiple back-end systems. Connector developers define what level of transaction support they need—none, local (with a single back-end system and its resource manager), or the other end of the spectrum, XA—with either single or two-phase commit for working across multiple back-end systems and their associated resource managers.
- A *security contract* enables a secure access to an EIS and provides support for a secure application environment, which reduces security threats to the EIS and protects valuable information resources managed by the EIS. The service enables the developer to define security between the EJB server and the back-end system. The specific security mechanism that is used is dependent on the security mechanism provided by the back-end system. For example, if a system requires Kerberos, the connection developer will include it. Under the contract, the connector provider must also support user authentication and any specific security contracts required by the back-end system.

In JEE 1.4, a new version (1.5) of the JCA was introduced and there are additional contracts that a resource adapter must support to handle new functionality and features in the JCA 1.5 specification. A resource adapter can support these four new contracts by implementing the required interfaces defined in the specification for each contract. These new contracts cover missing capabilities in the JCA 1.0 release. They are as follows:

- **Life Cycle Management Contract** The application server manages the startup and shutdown of the resource adapter.
- **Work Management Contract** Allows the resource adapter to do work by submitting it to an application server for execution. Since the application server does the work for the resource adapter, the resource adapter needn't worry about thread management. Instead, the application server manages this aspect efficiently and can use thread pooling if necessary.
- **Transaction Inflow Contract** This allows a resource adapter to propagate an imported transaction to an application server, as well as flow-in transaction completion and failure recovery initiated by an EIS.
- **Message Inflow Contract** This allows the resource adapter to synchronously or asynchronously deliver messages to end points in the application server, irrespective of message style, semantics, and infrastructure. In this way, different message providers, such as the Java Message Service (JMS) and Java API for XML Messaging (JAXM), can be plugged into J2EE application servers. This contract allows the EIS to be an active process, generating its own events and messages rather than a passive data source.

Common Client Interface

The JCA also defines a CCI for EIS access. The CCI defines a client API that is a standard for application components. It is analogous to the JDBC API standard. The CCI enables application components and EAI frameworks to drive interactions across heterogeneous EISs using a common client API. The CCI is intended as a standard for use by EAI and tools vendors.

Java Connector Architecture

With that background in mind, let's consider how the JCA specification as well as JEE in general compare to some of the features found in EAI vendors' products. Many EAI vendors, Tibco for example, have JCA support or are in the process of releasing products that incorporate JCA-based adapters. In light of this, and before we can discuss how JCA fits into the EAI picture, it's important that you first understand some basic EAI features:

- Resource adapters
- Data mapping
- Messaging brokers

Typical EAI vendors include proprietary adapters built to work with their products. These adapters allow for synchronous and asynchronous communication to an EIS. JCA adapters resemble those adapters, except JCA 1.0 adapters include only a synchronous communication channel. Resource adapters represent the EAI feature JCA most directly matches, although most EAI vendors' adapters offer more features than JCA adapters— for example, asynchronous capability. Note, JCA 1.5 introduces asynchronous communication via the Message Inflow Contract. JCA is young, but just like JDBC, it is a standard and when it matures it will be more desirable than having to maintain in-house domain knowledge on the plethora of vendor EAI software.

Data mapping means that data acquired in one format (for instance in the EIS's native format—such as an EBCDIC [extended binary code format] byte stream) by the resource adapter may have to be transformed into the format required for the business object. Mapping data from one system to another is time consuming because you must map each business object in both systems. In response, EAI vendors provide visual tools to enable a developer to set up such mapping. While JCA does not offer a data-mapping facility, EJB container-managed persistence (CMP) facility provides similar functionality. However, currently not all EJB containers can use EJB CMP with JCA. This will change as JCA use increases.

Messaging brokers, another feature common in many EAI products, usually enable both point-to-point (PTP) and publish/subscribe messaging. EAI products employ messaging as the connectivity layer to tie together disparate systems. JCA 1.0 does not address connectivity to an EIS in a message-oriented manner. It is possible, however, to implement some of a message broker's feature set by using JEE's Java Message Service (JMS). Here again, the new JCA 1.5 specification solves this deficiency, as it includes an asynchronous communication via the Message Inflow Contract.

JEE Connector Architecture: A General Integration Strategy

The majority of the work developers do today lies in creating new systems that must integrate with other systems. Integration can be simple to conceive but hard to accomplish; you can look at it in two ways:

- **Inbound integration** Outside systems initiate data requests to your system.
- **Outbound integration** Your system initiates data requests to other systems.

All of the following integration types are applicable in an inbound and an outbound manner. User interface (UI) integration, or "screen scraping" as it is known, represents a coarse type of integration. With UI-level integration, the data passed between systems will exist in the form of a UI representation. An outbound integration at the UI level entails requesting the UI as perhaps a web page from a remote system, and then possibly manipulating it before displaying it as if it were part of your system's UI. An inbound integration at the UI level entails allowing an outside system to request UI pages from your system for inclusion on a remote system. Note: You should choose UI integration over other options when it is unimportant to distinguish the data type being retrieved. UI integration often requires the least amount of effort to implement. UI integration is also least likely to scale well, because the original system may not be able to handle the load inflicted on it by a heavily used JEE application.

Message-level integration is growing in popularity, especially with the more widespread use of web services. It implies that the data passed between systems will be in the form of a message (a defined, data-driven text format). Outbound message integration involves requesting data from a remote system in a message form—for example, a SOAP (Simple Object Access Protocol) message. With an inbound integration, your system receives a request for data via a message and responds with a message. Message-oriented integration lends itself to loose coupling between systems because the systems remain unaware of the object types that exist on the remote system. That type of loose coupling works well with applications that wish to communicate over the Internet.

Object or RPC (remote procedure call) integration implies integrating systems using distributed objects (that is, using EJB calls to integrate). With object-level integration, data passes between systems as parameters to method calls. In an outbound object-level integration, your system invokes objects on remote systems, while in an inbound object-level integration, a remote system calls objects on your system to retrieve data. One of the main advantages of object-level integration is that you can call detailed APIs with full type safety and easily propagate the error codes and exceptions between systems.

Data-level integration implies that the data passed between systems will be in a data/record-oriented manner. In an outbound data-level integration, your system requests data in a record-oriented fashion from other systems. With an inbound data-level integration, a remote system requests data from your system in a record-oriented manner. The advantage of a data-level integration is that it lends itself to data mapping from one system onto the business objects in another system.

The Structure of the JCA

As mentioned, JCA's main components include the resource adapter, system contracts, and the CCI, which together give JCA the ability to access data in enterprise systems.

Resource Adapters and System Contracts

To use JCA in a JEE container, you must have a JCA resource adapter, which resembles a JDBC driver. A JCA adapter is specific to an EIS (for example, Tibco) and is contained in a Resource Adapter Archive (RAR) file composed of the JAR files and native libraries necessary to deploy the resource adapter on a JEE container. A JCA adapter interacts with a JEE server via system contracts. They enable the JEE server to propagate the context in which a JCA adapter is being called.

There are seven types of system contracts (three existed in JCA 1.0):

- Connection management
- Transaction management
- Security
- Life Cycle Management (JCA 1.5)
- Work Management Contracts (JCA 1.5)
- Message Inflow (JCA 1.5)
- Transaction Inflow Contracts (JCA 1.5)

Connection Management The connection management contract describes the agreement a JEE container has with the adapter regarding establishing, pooling, and tearing down connections. This contract allows listeners created on a connection to respond to events. (Also note that the underlying protocol an adapter uses to connect to an EIS is outside the scope of the JCA specification.)

JCA resource adapters must supply two implementations with the adapter. First, a *ConnectionFactory* provides a vehicle for creating connections. Second, the *Connection* class represents this particular resource adapter's underlying connection.

Transaction Management The transaction management contract controls transactions in two different ways. First, it allows distributed transactions that provide a mechanism to propagate transactions that originate from inside an application server to an EIS. For example, in an EJB, a transaction may be created. If this EJB then employs a JCA resource adapter, the transaction management contract enables the transaction to propagate to the EIS. In that circumstance, the transaction manager on the application server would control multiple resources to conduct distributed transaction coordination—for example, a two-phase commit.

In the second way, the transaction management contract can control transactions by creating *local transactions*. Local transactions are local in the sense that they exist only on a particular EIS resource. The contract provides transactions control, but they are related to any transaction that exists on the application server where the JCA resource adapter is running. Note that the resource adapter need not implement the transaction management contract. Making this optional allows for resource adapters in nontransaction resources.

Security The security contract enables the application server to connect to an EIS using security properties. The application server authenticates with the EIS by using security properties composed of a principal (a user ID) and credentials (a password). An application server can employ two methods to authenticate to an EIS (via a resource adapter).

With the first method, container-managed sign-on, the security credentials configure when the resource adapter is deployed on the application server. You can choose from several ways to configure security properties when using container-managed sign-on:

- **Configured identity** All resource adapter connections use the same identity when connecting to the EIS.
- **Principal mapping** The principal used when connecting to the EIS is based on a combination of the current principal in the application server and the mapping.
- **Caller impersonation** The principal used in the EIS exactly matches the principal in the application server.
- **Credentials mapping** Similar to caller impersonation, except the type of credentials must be mapped from application server credentials to EIS credentials.

While it's simple to configure the security properties at deployment time, this strategy proves less flexible because the security properties cannot change at runtime. Alternatively, you can configure security properties by component-managed sign-on, which allows you to pass security properties each time a connection is acquired from the resource adapter.

Life Cycle Management Contracts The *ResourceAdapter* interface in the *javax.resource.spi* package represents a resource adapter. There are two methods in the *ResourceAdapter* interface that allow for life cycle management: `start()` and `stop()`. The `start()` method is called when an application server wants to start a resource adapter (for example, to deploy it). The `stop()` method is called when the application server wants to release a resource adapter (for example, to undeploy it).

Work Management Contracts The Work Management contract allows the resource adapter to submit work to the application server. It does this by creating an object that extends the *Work* interface in the *javax.resource.spi.work* package. The *Work* interface is an extension of the *Runnable* interface. In addition to the `run()` method that it inherits from *Runnable* and which executes in its own thread, the *Work* interface contains a `release()` method.

Message Inflow and Transaction Inflow Contracts The Message Inflow contract allows the resource adapter to react to calls made by the application server to activate and deactivate message endpoints. The `endpointActivation()` method in the *ResourceAdapter* interface is called during endpoint activation. This causes the resource adapter to do the necessary setup for message delivery to the message endpoint. The `endpointDeactivation()` method of *ResourceAdapter* is called when a message endpoint is deactivated. This stops the resource adapter from delivering messages to the message endpoint. A `MessageEndpointFactory` object in the *javax.. resource.spi.endpoint* package is passed in to the `endpointActivation()` method. The object is used by the resource adapter to create a number of message endpoints. Any information about these endpoints should be removed from the resource adapter when the `endpointDeactivation()` method is called. Finally, the `getXAResources()` method of *ResourceAdapter* can be used to retrieve transaction resources in the event of a system crash. The `endpointActivation()`, `endpointDeactivation()`, and `getXAResources()` methods are mandated by the *ResourceAdapter* interface.

Common Client Interface

To retrieve and update data, you employ JCAs CCI layer, a method set resembling the type of commands used in JDBC to call a stored procedure. A JCA resource adapter is not required to support the CCI layer (the resource adapter creators can choose their own API set), and even if the resource adapter does support CCI, it may also support an API specific for that particular adapter. This owes to the diverse functionality contained in EIS software. Database operations boil down to add, update, delete, and inquire. EIS software may be more process oriented and hence would include a larger set of functionality.

Just like the JDBC API, the CCI APIs can be divided into four sections:

- APIs related to establishing a connection to an EIS, also referred to as the *connection interfaces*
- CCI APIs, which cover command execution on an EIS, referred to as the *interaction interfaces*
- *Record/ResultSet interfaces*, which encapsulate the query results to an EIS
- *Metadata interfaces*, which make it possible to examine an EISs metadata—for example, the attributes or type of EIS data to be queried

Code example illustrates JCA code to retrieve and update data

```
Public class JCAclass1...
int count;
try {
// obtain the connection

ConnectionSpec spec = new CciConnectionSpec(user, password);
Connection con = cf.getConnection(spec);
Interaction ix = con.createInteraction();
CciInteractionSpec iSpec = new CciInteractionSpec();

// command execution
iSpec.setSchema(user);
iSpec.setFunctionName("CLIENTCOUNT");

// handle the result set
RecordFactory rf = cf.getRecordFactory();
IndexedRecord iRec = rf.createIndexedRecord("InputRecord");
Record rec = ix.execute(iSpec, iRec);
Iterator iter = ((IndexedRecord)rec).iterator();
while(iter.hasNext())

{
   Object obj = iter.next();
if(obj instanceof Integer) count = ((Integer)obj).intValue();
}
// close the connection
con.close();
}
catch(Exception e)
 {
  e.printStackTrace();
 }
System.out.println("the count is  " + count);
...
```

Basic JCA 1.0 Adapter Implementation

Let's quickly explore the steps required to implement a JCA adapter—that is, a set of classes with which a JEE application server targets a particular enterprise system. As mentioned, a JCA adapter functions in much the same way as a JDBC driver connects to databases. We will describe the adapter's capabilities, as well as how to deploy and run it. We will see what occurs when the adapter executes in the container. It's important to frame the sample-adapter discussion by describing its functionality. Typically, when using an adapter we need to

- Determine the status of resources within the life cycle
- Establish a connection to the resource(s)
- Manage a transaction involving the resource
- Submit work to the resource
- Provide messaging capability to alert interested processes
- Provide security to protect resources

To use a JCA adapter, you need a JEE application server with JCA specification support. For example, BEA WebLogic version 8.1 supports JCA 1.0, and Weblogic version 9.1 supports JCA 1.5. You upload the resource adapter archive—for example, the *Ucnyadapter.rar* file. The adapter includes two class categories:

- **Managed classes** The application server calls managed classes to perform the connection management. They're needed only if the application server is managing the connection via a connection pool, which is probably the case.
- **Physical connection classes** These required classes, which the aforementioned managed classes may call, establish the connection to the EIS.

ManagedConnectionFactory

With the *UCManagedConnectionFactory* class, which implements the *ManagedConnectionFactory* interface, you create the *UCConnectionFactory* and *UCManagedConnection* classes. In JCA 1.5, the *ManagedConnectionFactory* interface remains unchanged to preserve backward compatibility, but if you want outbound resources to have access to the capabilities provided to the resource adapter, also implement the new *ResourceAdapterAssociation* interface. The *UCManagedConnectionFactory* class acts as the main entry point for the application server to call into the adapter:

```
package ucnyjca;

import java.io.PrintWriter;
import java.io.Serializable;
import java.sql.DriverManager;
import java.util.Iterator;
import java.util.Set;
import javax.resource.ResourceException;
import javax.resource.spi.*;
import javax.security.auth.Subject;

public class UCManagedConnectionFactory
implements ManagedConnectionFactory, Serializable
{

    public UCManagedConnectionFactory() {
        System.out.println("We are executing

UCManagedConnectionFactory.constructor");
    }

    public Object createConnectionFactory(ConnectionManager cxManager)

throws ResourceException {
        System.out.println("We are executing
UCManagedConnectionFactory.createConnectionFactory,1");
        return new UCDataSource(this, cxManager);
    }

    public Object createConnectionFactory() throws ResourceException {

              System.out.println("We are executing
UCManagedConnectionFactory.createManagedFactory,2");
        return new UCDataSource(this, null);
    }

    public ManagedConnection createManagedConnection
(Subject subject, ConnectionRequestInfo info) {
        System.out.println("We are executing
UCManagedConnectionFactory.createManagedConnection");
        return new UCManagedConnection(this, "test");
    }
    public ManagedConnection matchManagedConnections
(Set connectionSet, Subject subject, ConnectionRequestInfo info)
        throws ResourceException
    {
        System.out.println("We are executing
UCManagedConnectionFactory.matchManagedConnections");
        return null;
    }

    public void setLogWriter(PrintWriter out) throws ResourceException {
        System.out.println("We are executing
UCManagedConnectionFactory.setLogWriter");
    }

    public PrintWriter getLogWriter() throws ResourceException {
        System.out.println("We are executing

UCManagedConnectionFactory.getLogWriter");
        return DriverManager.getLogWriter();
    }

    public boolean equals(Object obj) {
        if(obj == null)
            return false;
        if(obj instanceof UCManagedConnectionFactory)
        {
            int hash1 = ((UCManagedConnectionFactory)obj).hashCode();
            int hash2 = hashCode();
            return hash1 == hash2;
        }
        else
        {
            return false;
        }
    }

    public int hashCode()
    {
              return 1;
    }
}
```

ManagedConnection

The *UCManagedConnection* class implements the *ManagedConnection* interface. *UCManagedConnection* encapsulates the adapter's physical connection, in this case the *UCConnection* class:

```
package ucnyjca;

import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.*;
import javax.resource.NotSupportedException;
import javax.resource.ResourceException;
import javax.resource.spi.*;
import javax.security.auth.Subject;
import javax.transaction.xa.XAResource;


public class UCManagedConnection
  implements ManagedConnection
{
     private UCConnectionEventListener UCListener;
     private String user;
     private ManagedConnectionFactory mcf;
     private PrintWriter logWriter;
     private boolean destroyed;
     private Set connectionSet;

  UCManagedConnection(ManagedConnectionFactory mcf, String user)
  {
    System.out.println("We are executing UCManagedConnection");
    this.mcf = mcf;
    this.user = user;
    connectionSet = new HashSet();
    UCListener = new UCConnectionEventListener(this);
  }

  private void throwResourceException(SQLException ex)
    throws ResourceException
  {
    ResourceException re = new ResourceException("SQLException: " +
ex.getMessage());
    re.setLinkedException(ex);
    throw re;
  }
  public Object getConnection(Subject subject, ConnectionRequestInfo
connectionRequestInfo)
    throws ResourceException
  {
    System.out.println("We are executing UCManagedConnection.getConnection");
    UCConnection UCCon = new UCConnection(this);
    addUCConnection(UCCon);
    return UCCon;
  }

  public void destroy()
  {
        System.out.println("We are executing UCManagedConnection.destroy");
        destroyed = true;
  }

  public void cleanup()
   {
        System.out.println("We are executing UCManagedConnection.cleanup");
  }

  public void associateConnection(Object connection)
   {
        System.out.println("We are executing
UCManagedConnection.associateConnection");
  }

  public void addConnectionEventListener(ConnectionEventListener listener)
  {
        System.out.println("We are executing
UCManagedConnection.addConnectionEventListener");
    UCListener.addConnectorListener(listener);
  }

  public void removeConnectionEventListener(ConnectionEventListener listener)
  {
        System.out.println("We are executing
UCManagedConnection.removeConnectionEventListener");
    UCListener.removeConnectorListener(listener);
  }

  public XAResource getXAResource()
    throws ResourceException
  {
        System.out.println("We are executing
UCManagedConnection.getXAResource");
    return null;
  }

  public LocalTransaction getLocalTransaction()
  {
           System.out.println("We are executing
UCManagedConnection.getLocalTransaction");
      return null;
  }

  public ManagedConnectionMetaData getMetaData()
    throws ResourceException
  {
    System.out.println("We are executing UCManagedConnection.getMetaData");
    return new UCConnectionMetaData(this);
  }


  public void setLogWriter(PrintWriter out)
    throws ResourceException
  {
        System.out.println("We are executing UCManagedConnection.setLogWriter");
    logWriter = out;
  }

  public PrintWriter getLogWriter()
    throws ResourceException
  {
        System.out.println("We are executing UCManagedConnection.getLogWriter");
    return logWriter;
  }

  Connection getUCConnection()
    throws ResourceException
  {
        System.out.println("We are executing
UCManagedConnection.getUCConnection");
    return null;
  }

  boolean isDestroyed()
  {
        System.out.println("We are executing UCManagedConnection.isDestroyed");
    return destroyed;
  }

  String getUserName()
  {
        System.out.println("We are executing UCManagedConnection.getUserName");
    return user;
  }

  void sendEvent(int eventType, Exception ex)
  {
        System.out.println("We are executing UCManagedConnection.sendEvent,1");
    UCListener.sendEvent(eventType, ex, null);
  }

  void sendEvent(int eventType, Exception ex, Object connectionHandle)
  {
        System.out.println("We are executing UCManagedConnection.sendEvent,2 ");
    UCListener.sendEvent(eventType, ex, connectionHandle);
  }

  void removeUCConnection(UCConnection UCCon)
  {
        System.out.println("We are executing
UCManagedConnection.removeUCConnection");
    connectionSet.remove(UCCon);
  }

  void addUCConnection(UCConnection UCCon)
  {
        System.out.println("We are executing
UCManagedConnection.addUCConnection");
    connectionSet.add(UCCon);
  }

  ManagedConnectionFactory getManagedConnectionFactory()
  {
        System.out.println("We are executing
UCManagedConnection.getManagedConnectionFactory");
    return mcf;
  }

}
```

UCConnectionEventListener

The *UCConnectionEventListener* class allows the application server to register callbacks for the adapter. The application server can then perform operations—connection-pool maintenance, for example—based on the connection state:

```
package ucnyjca;

import java.util.Vector;
import javax.resource.spi.ConnectionEvent;
import javax.resource.spi.ConnectionEventListener;
import javax.resource.spi.ManagedConnection;

public class UCConnectionEventListener
    implements javax.sql.ConnectionEventListener
{
    private Vector listeners;
    private ManagedConnection mcon;


    public UCConnectionEventListener(ManagedConnection mcon)
    {
        System.out.println("We are executing UCConnectionEventListener");
        this.mcon = mcon;
    }

    public void sendEvent(int eventType, Exception ex, Object connectionHandle)
    {
        System.out.println("We are executing
UCConnectionEventListener.sendEvent");
    }

    public void addConnectorListener(ConnectionEventListener l)
    {
        System.out.println("We are executing
UCConnectionEventListener.addConnectorListener");
    }

    public void removeConnectorListener(ConnectionEventListener l)
    {
        System.out.println("We are executing
UCConnectionEventListener.removeConnectorListener");
    }

    public void connectionClosed(javax.sql.ConnectionEvent connectionevent)
    {
        System.out.println("We are executing
UCConnectionEventListener.connectorClosed");
    }
    public void connectionErrorOccurred(javax.sql.ConnectionEvent event)
    {
        System.out.println("We are executing
UCConnectionEventListener.connectorErrorOccurred");
    }

}
```

UCConnectionMetaData

The *UCConnectionMetaData* class provides meta-information—for example, the maximum number of connections allowed, and so on—regarding the managed connection and the underlying physical connection class:

```
package ucnyjca;

import javax.resource.ResourceException;
import javax.resource.spi.*;


public class UCConnectionMetaData
    implements ManagedConnectionMetaData
{

    private UCManagedConnection mc;

    public UCConnectionMetaData(UCManagedConnection mc)
    {
        System.out.println("We are executing UCConnectionMetaData.constructor");
        this.mc = mc;
    }

    public String getEISProductName()
        throws ResourceException
    {
        System.out.println("We are executing
UCConnectionMetaData.getEISProductName");
        return "ucnyjca";
    }

    public String getEISProductVersion()
        throws ResourceException
    {
        System.out.println("We are executing
  UCConnectionMetaData.getEISProductVersion");
       return "1.0";

    }

    public int getMaxConnections()
        throws ResourceException
    {
            System.out.println("We are executing

  UCConnectionMetaData.getMaxConnections");
            return 5;
    }

    public String getUserName()
        throws ResourceException
    {
           return mc.getUserName();
    }

}
```

UCConnection

The *UCConnection* class represents the "handle" to the underlying physical connection to the EIS. *UCConnection* is one of the few classes that does not implement an interface in the JCA specification. The implementation that follows is simple, but a working implementation might contain connectivity code using sockets, as well as other functionality:

```
package ucnyjca;

public class UCConnection
{
    private UCManagedConnection mc;

    public UCConnection(UCManagedConnection mc)
    {
        System.out.println("We are executing UCConnection");
        this.mc = mc;
    }
}
```

UCConnectionRequestInfo

The *UCConnectionRequestInfo* class contains the data (such as the username, password, and other information) necessary to establish a connection:

```
package ucnyjca;

import javax.resource.spi.ConnectionRequestInfo;

public class UCConnectionRequestInfo
    implements ConnectionRequestInfo
{

    private String user;
    private String password;

    public UCConnectionRequestInfo(String user, String password)
    {
        System.out.println("We are executing UCConnectionRequestInfo");
        this.user = user;
        this.password = password;
    }

    public String getUser()
    {
        System.out.println("We are executing UCConnectionRequestInfo.getUser");
        return user;
     }

    public String getPassword()
    {
        System.out.println("We are executing
UCConnectionRequestInfo.getPassword");
        return password;
    }

    public boolean equals(Object obj)
    {
        System.out.println("We are executing UCConnectionRequestInfo.equals");
        if(obj == null)
            return false;
        if(obj instanceof UCConnectionRequestInfo)
        {
            UCConnectionRequestInfo other = (UCConnectionRequestInfo)obj;
            return isEqual(user, other.user) &&
isEqual(password, other.password);
        } else
        {
            return false;
        }
    }

    public int hashCode()
    {
        System.out.println("We are executing UCConnectionRequestInfo.hashCode");
        String result = "" + user + password;
        return result.hashCode();
    }

    private boolean isEqual(Object o1, Object o2)
    {
        System.out.println("We are executing UCConnectionRequestInfo.isEqual");
        if(o1 == null)
            return o2 == null;
        else
            return o1.equals(o2);
    }

}
```

UCDataSource

The *UCDataSource* class serves as a connection factory for the underlying connections. Because the example adapter does not implement the CCI interfaces, it implements the *DataSource* interface in the *javax.sql package*:

```
package ucnyjca;

import java.io.PrintWriter;
import java.io.Serializable;
import java.sql.*;
import javax.naming.Reference;
import javax.resource.Referenceable;
import javax.resource.ResourceException;
import javax.resource.spi.ConnectionManager;
import javax.resource.spi.ManagedConnectionFactory;
import javax.sql.DataSource;
public class UCDataSource
    implements DataSource, Serializable, Referenceable
```

```
{
    private String desc;
    private ManagedConnectionFactory mcf;
    private ConnectionManager cm;
    private Reference reference;
    public UCDataSource(ManagedConnectionFactory mcf, ConnectionManager cm)
    {
        System.out.println("We are executing UCDataSource");
        this.mcf = mcf;
        if(cm == null)
            this.cm = new UCConnectionManager();
        else
            this.cm = cm;
    }
        public Connection getConnection(String username, String password)
        throws SQLException
    {
        System.out.println("We are executing UCDataSource.getConnection,2");
        try
        {
            javax.resource.spi.ConnectionRequestInfo
info = new UCConnectionRequestInfo(username, password);
            return (Connection)cm.allocateConnection(mcf, info);
        }
        catch(ResourceException ex)
        {
            throw new SQLException(ex.getMessage());
        }
    }

    public int getLoginTimeout()
        throws SQLException
    {
        return DriverManager.getLoginTimeout();
    }

    public void setLoginTimeout(int seconds)
        throws SQLException
    {
        DriverManager.setLoginTimeout(seconds);
    }

    public PrintWriter getLogWriter()
        throws SQLException
    {
        return DriverManager.getLogWriter();
    }

    public void setLogWriter(PrintWriter out)
        throws SQLException
    {
        DriverManager.setLogWriter(out);
    }

    public String getDescription()
    {
        return desc;
    }

    public void setDescription(String desc)
    {
        this.desc = desc;
    }

    public void setReference(Reference reference)
    {
        this.reference = reference;
    }

    public Reference getReference()
    {
        return reference;
    }
}
```

JCA 1.5 Adapter Implementation

Here, we will cover JCA 1.5 enhancements to provide you with a background for how they fit into the overall architecture of JEE / EIS connectivity. Here again, the four new contracts are as follows:

- Life Cycle Management
- Work Management Contracts
- Message Inflow
- Transaction Inflow Contracts

The JCA 1.5 specification defines resource adapters as: "a system-level software driver that is used by a Java application to connect to an EIS. The resource adapter plugs into an application server and provides connectivity between the EIS, the application server, and the enterprise application."

With respect to Life Cycle Management, the JCA 1.0 release provided a life cycle model for managed connections and their associated connection handles. It included no such life cycle for the resource adapter. The first moment of life for the deployed resource adapter was when a managed-connection factory was created. This has been corrected in JCA 1.5 . The *ResourceAdapter* interface in the *javax.resource.spi* package represents a resource adapter. There are two methods in the *ResourceAdapter* interface that control life cycle management: `start()` and `stop()`. The `start()` method is used when an application server starts a resource adapter . The `stop()` method is used by the application server to release a resource adapter. At startup, the application server creates an instance of the <resourceadpter-class> and sets the properties as specified in the deployment descriptor. The `start` method is then called, passing in an object implementing the *BootstrapContext* interface. You can use this object to: create timers to be used to schedule work at a specified interval or datetime, schedule work on other threads, and control imported transactions. The application server typically calls the `stop()` method on the resource adapter as part of the shutdown process. The JCA 1.5 specification describes two phases to this process. First, the application server ensures that all applications that depend on the resource adapter have stopped. The application server then calls the `stop()` method. The resource adapter then performs an housekeeping (for example, release resources) and ultimate shutdown.

The Work Management contract provides a way for the resource adapter to submit work to the application server. The *WorkManager* lets a resource adapter schedule work for synchronous or asynchronous execution on an application-server thread. This work can be performed in a transaction the resource adapter imports, in which case the *XATerminator* lets the work be completed. A `Timer` provides for delayed or periodic execution of work. It creates an object that extends the *Work* interface in the *javax.resource.spi.work* package. The *Work* interface extends *Runnable*, and you should implement the work to be performed in the `run()` method, similar to a program that uses Java threads directly. The *WorkManager* interface provides three sets of methods (`doWork`, `startWork`, and `scheduleWork`) for processing work.

The `doWork` methods on *WorkManager* allow work to be executed synchronously, blocking until that work has completed. If the application server is busy, it might defer the start of this piece of work. The `doWork()` method gives the application server more control over the work that a resource adapter performs. If the resource adapter is in the middle of a long, complicated operation when the application server is shutting down, then the server doesn't need to wait for the resource adapter to finish or pull the rug out from under its feet. Instead, the server can signal to the resource adapter by calling the `release()` method on the `Work` object. The resource adapter should then complete processing as quickly as possible. Also the `startWork()` and `scheduleWork()` methods on *WorkManager* let a resource adapter process work asynchronously while still keeping the application server in control. The `startWork()` method waits until the piece of work has started to execute but not until it completes. This method can therefore be used by a caller that needs to know that the work will be performed but does not need to wait until it has finished. The `scheduleWork()` method returns as soon as the work has been accepted for processing. In this case, there are no guarantees that work will actually be performed.

The Message Inflow contract provides the ability for the resource adapter to respond to calls made by the application server to activate and deactivate message endpoints. The `endpointActivation()` method in the *ResourceAdapter* interface is called to do the necessary setup for message delivery to the message endpoint. The `endpointDeactivation()` method of *ResourceAdapter* is called when a message endpoint is no longer needed, i.e., this stops the resource adapter from delivering messages. A `MessageEndpointFactory` object in the *javax.resource .spi.endpoint* package is passed in to the `endpointActivation()` method. The object is used by the resource adapter to create message endpoints. Any information about these endpoints should be removed from the resource adapter when the `endpointDeactivation()` method is called. The `getXAResources()` method of *ResourceAdapter* retrieves transaction resources in the event of a system failure. The `endpointActivation()`, `endpointDeactivation()`, and `getXAResources()` methods are mandatory.

Here is an example implementation of the *ResourceAdapter* interface that illustrates the life cycle management and work management contracts:

```
public interface BootstrapContext {


   WorkManager getWorkManager();

   XATerminator getXATerminator();

   Timer createTimer() throws UnavailableException;

}
```

```
public class UcnyResourceAdapterImpl implements
       ResourceAdapter {
  public static final long TIMEOUT_FIVE_SECONDS = 5000L;
```

Note if the application server is particularly busy, it might defer the start of this piece of work. You can use the second `startTimeout` parameter to specify how long the resource adapter is prepared to wait for the work to start. If the application server fails to start the work within this time then, again, a `WorkRejectedException` is thrown.

```
  public UcnyResourceAdapterImpl() {    }

  public void start(BootstrapContext ctx)
```

Notice that there is an object passed in with the `start()` method that implements the *BootstrapContext* interface. This object allows the EIS to pass transaction information to the application server, as well as the ability to pass work to the application server.

```
   throws ResourceAdapterInternalException
  {

      WorkManager workManager = ctx.getWorkManager();
      Work UcnyWorkJob = new UcnyWorkImpl();
      WorkListener workListener =
              new UcnyWorkListenerImpl();

      try {
```

With the xWork methods, the resource adapter can optionally pass a listener that will be notified as the item of work passes through the states of accepted, started, and completed or, in the failure case, rejected.

```
           workManager.startWork
                  (UcnyWorkJob, TIMEOUT_FIVE_SECONDS,
                  new ExecutionContext(), workListener);
```

Also, it is possible to have the piece of work executed in the context of a transaction imported by the resource adapter, rather than the context associated with the current thread. The third parameter is an optional `ExecutionContext`.

```
          } catch (WorkException e) {

             //  Handle the exception

         }

       }


      public void stop()

      {

         //  PERFORM HOUSKEEPING AND FINISH

      }



     }
public class UcnyWorkJob implements Work {


  void run() {

          // HERE IS WHERE THE WORK ACTUALLY TAKES PLACE       }

   }


   void release() {

       WHEN WE FINISH THE WORK WE RELEASE THE RESOURCES

         }

}
```

Typically, you want the application server to notify you about the status of submitted work that is in a partial state of execution. To accomplish the same in JCA 1.5, you create an object that implements the *javax.resource.spi.WorkListener* interface. You can then register this object using the `startWork()` method of the `WorkManager` object on the application server. The *WorkManager* interface facilitates the submission of `Work` instances for execution. Registering the object allows the server to notify the resource adapter if the work was rejected or accepted, and if accepted, when the work was started and completed. You can also extend the `WorkAdapter` class, which implements the *WorkListener* interface and provides empty methods for each of these. Here is skeleton code for a class that implements *WorkListener*:

```
public class UcnyWorkListenerImpl implements WorkListener {
```

This listener is used to provide notification back to the originator of a piece of work once it has completed, or to reschedule an item of work when it fails.

```
      public void workAccepted(WorkEvent e) {

         //  System.out.println("Work instance " + e +

         //      " has been accepted.");

     }


     public void workRejected(WorkEvent e) {

         //  System.out.println("Work instance " + e +

         //      " has been rejected.");

     }


     public void workStarted(WorkEvent e) {

         //  System.out.println("Work instance " + e +

         //      " has been started.");

     }
```

```
            public void workCompleted(WorkEvent e) {

                //  System.out.println("Work instance " + e +

                //      " has been completed.");
        }


        }
```

Here is an implementation of the *ResourceAdapter* interface that illustrates the JCA 1.5 life cycle management and work management contracts:

```
public class UcnyResourceAdapterImpl implements

ResourceAdapter {


// Lifecycle Contract methods from earlier omitted.


public XAResource[] getXAResources(ActivationSpec[] specs)

throws ResourceException

{

// return XAResource objects that correspond to ActivationSpecs passed


return null;

}


public void endpointActivation(MessageEndpointFactory mef,

ActivationSpec as)

throws NotSupportedException

{


}
public void endpointDeactivation(MessageEndpointFactory mef,

ActivationSpec as)

{

}


}
```

The `ActivationSpec` class that is passed in to the `ResourceAdapter` methods is a JavaBean that implements a number of get and set methods for various properties. In addition to providing these get and set methods, an implementation must also provide a `validate()` method to ensure that all of the properties have been legally set. If a property has not been set properly, the method must throw an `InvalidProp ertyException`. Note that an `ActivationSpec` object cannot override `equals()`.

```
public class MyActivationSpec implements ActivationSpec,

Serializable {


public void setMyProperty(MyProperty s) { }

public MyProperty getMyProperty() { }


public void validate() throws InvalidPropertyException { }


}
```

In version 1.0 of the J2EE Connector Architecture, a resource adapter could only pass transaction information to the EIS, either from itself or from an external transaction manager. However with the Transaction Inflow contract in version 1.5 of the architecture, the resource adapter can pass EIS transaction requests to the application server as well as use the `BootstrapContext` object that is passed in with the `start()` method of the Life Cycle Contract. The *BootStrapContext* interface was mentioned briefly in the discussion of the Life Cycle Management contract. Here are the methods in the *BootstrapContext* interface:

```
public class UcnyBootstrapContextImpl implements

BootstrapContext {
```

```
public WorkManager getWorkManager() {

// Get the work manager from the application server

}


public XATerminator getXATerminator() {

return new UcnyXATerminatorImpl();

}


public Timer createTimer() {

return new Timer();

}

}
```

Let's look at the *XATerminator* interface. Notice that it's the return type of the `getXATerminator()` method in the *BootStrapContext* interface. The *XATerminator* interface contains five simple methods that handle transactions:

```
public class UcnyXATerminatorImpl implements XATerminator {


public void commit(Xid xid, boolean onePhase)

throws XAException { }

public void forget(Xid xid) throws XAException { }

public int prepare(Xid xid) throws XAException { }

public Xid[] recover(int flag) throws XAException { }

public void rollback(Xid xid) throws XAException { }


}
```

Build the RAR File

The next step is to build the *ucnyjca.rar* file. Typically, you would have a source directory containing two subdirectories: *ucnyjca* containing the *.java* files, and *META-INF* containing the configuration files. See the JCA 1.5 connector specification located at *http://java.sun.com/j2ee/connector/download.html* for information on resource adapter deployment descriptors. The deployment descriptors, the *ra.xml* file needs to be in the *WEB-INF* directory of the WAR file. The resource adapter descriptor file, *ra.xml*, is fairly easy to create. You simply need to point in the file to the class that implements the *ResourceAdapter* interface. The application server will then access that class.

```
     <?xml version="1.0" encoding="UTF-8"?>

    <connector xmlns="http://java.sun.com/xml/ns/j2ee"

      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee

      http://java.sun.com/xml/ns/j2ee/connector_1_5.xsd"

      version="1.5">


      <display-name>UCNY Resource Adapter</display-name>

      <vendor-name>UCNY, Inc.</vendor-name>

      <eis-type>Unknown</eis-type>

      <resourceadapter-version>1.0</resourceadapter-version>


      <resourceadapter>

  <resourceadapter-class>

  com.Ucny.ra.UcnyResourceAdapterImpl

  </resourceadapter-class>

  </resourceadapter>


  </connector>
```

To compile and build the RAR file:

1. Compile the class files: *javac *.java* in the *ucnyjca* directory.
2. Build *ucnyjca.jar* :
   ```
   jar cvf ucnyjca.jar ucnyjca
   ```
3. Create the RAR file using *ucnyjca.jar* and the *META-INF* directory:
   ```
   jar cvf ucnyjca.rar ucnyjca.jar META-INF
   ```

The JCA specification's complexity makes implementing even basic adapters a difficult task. Moreover, the task grows when you add the transaction and security contracts (not implemented for this example), as well as the CCIs. The complexity shows that the JCA specification is oriented toward commercial software vendors implementing adapters and their customers using them. In this arena, the JCA makes sense, although a less flexible, simpler interface version would be nice. Therefore, if you are considering using JCA to connect to a legacy system in your enterprise, you would make the implementation more likely by leveraging an off-the-shelf adapter rather than developing your own. If the system to which you need connectivity does not have a JCA adapter, consider an alternative approach. Perhaps, the old style for a "work-around" or using web services may provide the best solution.

For its part, while JCA is still a new standard, it shows promise for making the task of integrating with an EIS less daunting. It is, however, still some distance off in terms of general availability for most vendors. The economy, when it eventually makes an upturn, will create opportunities for JEE architects to apply JCA to reface the enterprise.

CERTIFICATION OBJECTIVE 6.01

Distinguish Appropriate from Inappropriate Techniques for Providing Access to a Legacy System from Java Technology Code Given an Outline Description of That Legacy System

The following ten exercises are in the form of practice essay questions:

1. Read the question.
2. Develop an essay-style answer.

3. Review the draft and finalize your response.
4. Review the answer in the book.

EXERCISE 6-1

Techniques and Best Practices

Question As an enterprise architect who is commissioned to enable a set of existing legacy or EIS systems to handle JEE technology, what are some of the techniques and best practices that you might incorporate?

Answer When integrating existing EIS with any new technology, especially JEE, it's given that EIS integrates more easily when using proven guidelines and standards. Following are some tried-and-true concepts.

JEE systems that access existing or external information resources should avoid accessing those resources directly from multiple locations, i.e., use one point of access to facilitate changes and avoid potential data integrity problems. Otherwise, multiple access entwines your business logic with the implementation details of an external resource. If the API to that resource changes (when, for example, you change resource vendors, a new version is released, or for other reasons), changes to your JEE application source code will be necessary throughout the application, and the resulting testing burden can be considerable.

Per our desire to adhere to standards for some resources, an EIS resource vendor or some third party may provide a JEE connector extension, an adapter that allows JEE systems to interoperate with other EIS resources transparently and with transaction management capability.

For database access, a standard in the JDBC API makes vendors' proprietary technology accessible in an open way. Switching database implementations, even at runtime, is facilitated with JDBC, which provides a standard API to mask the vendor-specific implementation details in connection configuration data and access and transaction functionality.

In addition, it is a good practice to use a design pattern to encapsulate access to EIS resources and prepare for eventual migration to a JCA-based interface. If no JEE connector extension is available for your EIS resource, a good alternative is to use DAO classes to represent the EIS as an abstract resource. Instead of calling the EIS directly from the enterprise bean, create a DAO class that represents the services your bean needs. This is an application of the Bridge design pattern, which makes an interface's implementation transparently replaceable by decoupling the implementation from the interface. A DAO class that "wraps" an EIS resource insulates the enterprise bean from changes in that resource. New versions of the EIS resource can then be implemented and the change control will be the only necessary modifications to the DAO layer of the JEE application. Another benefit of this practice is experienced when a connector becomes available for your EIS resource. The enterprise can replace the existing DAO implementation with one that simply dispatches calls to the connector.

Imagine that the enterprise is using a custom legacy system to which your enterprise bean needs access. A DAO class can provide a "vendor-agnostic" API interface to the enterprise bean, while handling the details of service requests from the enterprise bean to the legacy system. This scheme is advantageous when a single service request from your JEE server's perspective requires access to a number of existing EIS resources—that is, a DAO can be used to facade multiple EIS resources. This use of the façade pattern facilitates changes to these EIS services. When an existing EIS service is replaced, the existing DAO class can be replaced with a new DAO class that presents the new service to the enterprise bean in terms of the existing DAO interface. Isolating your enterprise bean functionality with a DAO layer makes it easier for your JEE system design to evolve with time.

The DAOclass(es) should reflect the functional requirements of the services your enterprise beans need, not necessarily the structure of the existing system. A DAO class' interface should reflect a current view. Analyze what the existing EIS does, determine what needs to be done today and tomorrow, and create methods in the DAO classes that provide the most frequently required functionality. If multiple EIS resources are required to perform a single task, the DAO class can combine access to these systems and present them to the EJB as a single service. So as an EIS integration "best practice," we should avoid letting the structure of existing EIS resources dictate the structure of the integrated system. Instead, architect and design with your new requirements and goals in mind. Use existing legacy resources as services to meet those requirements.

A DAO class should be neither a collection of unrelated tools nor a tool designed for one application, but something that cleanly and completely represents a clear and reusable abstraction. UML diagrams such as collaboration, state chart, activity, and package diagrams can be a help in the analysis.

EXERCISE 6-2

Implementing Data Validation and Referential Integrity Contraints

These essay questions will help develop the ability to articulate and describe the JEE concepts and components used in parts 2 and 3 of the exam.

Question As an architect integrating a JEE system with an existing EIS database system, where should data validation and referential integrity constraints be implemented?

Answer This is a difficult call. The practical aspects of the decision revolve around the following:

- How much will it ultimately cost?
- How much is already invested in the database application?
- How long is it expected to be functional?

If the DBMS is relational and were implemented after the mid 1980s, it is typically best to use the DBMS functionality to enforce value and referential integrity. Sybase, Oracle, SQL Server, DB2, and Informix—to mention the most popular DBMSs—have had these abilities for many years. These DBMSs include declarative value and referential integrity constraint features, integrated with the Data Description Language (DDL), and they provide built-in declarative triggers to handle cascading actions required for referential integrity, such as deleting all item rows in a canceled order. Implementing these in the enterprise bean layer would duplicate logic, making maintenance difficult. Any change to the database constraints would require making the change to enterprise beans and to the database.

The architectural benefits and capabilities maintaining data integrity constraints in the database layer include the following:

- **Facilitated use by multiple applications** If for some reason multiple applications are responsible for maintaining database integrity, every application creates an opportunity for bugs that would violate that integrity. Furthermore, other applications that may want to access the database are relieved of the duty of maintaining integrity constraints. They still must, of course, deal with error conditions that result if they violate those constraints.

- **Centralization** If the constraints are maintained only in the database, the database is the one place where data can be considered consistent by definition. If data inconsistencies exist, either the integrity constraints are incorrect or the design has flaws.
- **Portability** Simple value and integrity constraints, such as primary keys, simple foreign keys, uniqueness, value range checking, and so on, are reasonably portable.
- **Performance and reliability** Database vendors that offer database constraints features have invested a great deal of time and money in ensuring that those features operate correctly and efficiently.

The drawbacks of using the DBMS built-in database integrity constraints mechanisms and the EJB can include the following:

- **Possible duplication of logic** Enterprise beans generally need reasonable data to perform properly. Therefore, most well-designed enterprise beans do a reasonably good job of checking data values and existence constraints. Database integrity violation errors usually indicate a bug or a problem with the design. Nevertheless, the logic enforcing value and referential integrity is necessarily duplicated. Changing the integrity rules in the database will usually also entail changes to the code, and keeping the two synchronized can be a problem.
- **Potential nonportability of DBMS constraints** While simple value and referential integrity constraints are fairly portable, databases differ in coverage and syntax for more involved mechanisms such as composite foreign keys, database triggers, and procedural triggers. Procedural triggers in particular are portability concerns, because, when offered, they are often written in the database vendor's product-specific proprietary language. For example, Sybase Transact SQL is very different from the Oracle PL/SQL procedure language.
- **Database definition and configuration is uncontrolled** Because database constraint and trigger configuration are performed with the database vendor's tools, such constraints are maintained outside of the JEE server framework. Because the data model constraints are specified not in the deployment descriptor but in the persistence layer, such constraints are not part of a JEE server deployment. They must therefore be managed separately, complicating deployment and maintenance and providing another possible avenue for system flaws.

Another option is to use the EJB to handle constraints. Referential integrity constraints can be implemented in the EJB tier. The constraints required for an application may not be available in the DBMS chosen. The data model may have constraint requirements that cannot be satisfied using the DBMS constraint language. Such constraints can be implemented in the EJB tier. EJB-tier constraint management also provides portability, since the enterprise beans will operate identically in JEE-branded containers. Constraints in the EJB tier can also be controlled by way of environment settings in the application deployment descriptor, centralizing constraint management and making it controllable at deploy time.

Yet another option is to implement constraints in both the EJB and database tiers and configuring the constraint implementation at deploy time. This strategy is useful especially when an application must be portable to many different databases, and you want consistent behavior across vendors while optimizing performance by using each database's full power.

You could also create a persistence server for the EIS tier. Constraints should be expressed in a declarative constraint language provided by the database vendor. In their absence, the implementation should choose to wrap a layer of integrity management software around the database API. The EIS tier of your application can be an API that you create to wrap the database. Your application accesses the data store only through that server. This application of the decorator design pattern can provide a solution that is portable across databases, is declaratively configurable, and provides a consistent behavior across various clients. As a great deal of design, construction, validation, and maintenance are required, it should be the solution where ultimate flexibility and portability is required.

Finally, commercial transaction processing (TP) monitors provide the benefits of the persistence server just described. TP monitors can provide scalability and availability. Typically, they work with multiple database vendors. You avoid vendor "lock-in" by wrapping calls to the monitor in DAO classes.

EXERCISE 6-3

Legacy Mapping

Question What is legacy object mapping?

Answer Legacy object mapping builds wrappers around legacy system interfaces to access elements of the legacy business logic and database tiers directly. Legacy object mapping tools are used to create proxy objects that access legacy system functions and make them available in an object-oriented manner.

EXERCISE 6-4

Transaction Monitors

Question What is the purpose of a transaction monitor?

Answer Transaction monitors are programs, such as IBM CICS, that monitor transactions, to ensure that they are completed in a successful manner. They ensure that successful transactions are committed, that unsuccessful transactions are aborted, and that the in-flight data updates are rolled back to the status quo ante or the state it was before the attempted change.

EXERCISE 6-5

Off-Board Servers

Question What is an off-board server?

Answer An off-board server is a server that executes as a proxy for a legacy system. It communicates with the legacy system using the custom protocols supported by the legacy system. It communicates with external applications using industry-standard protocols.

EXERCISE 6-6

JDBC vs. ODBC

Question How does Java Database Connectivity (JDBC) differ from the Microsoft database connectivity interface (Open Database Connectivity, or ODBC)?

Answer ODBC is the industry-standard interface by which database clients connect to database servers. JDBC is a pure Java solution that does not follow the ODBC standard. A bridge between JDBC and ODBC allows JDBC to access databases that support ODBC.

EXERCISE 6-7

Accessing Legacy System Software

Question How is Java Native Interface (JNI) used to access legacy system software?

Answer JNI is used to write custom code to interface Java objects with legacy software that does not support standard communication interfaces.

EXERCISE 6-8

Accessing COM Objects

Question How is Java-to-COM bridging used to access COM objects?

Answer A Java-to-COM bridge enables COM objects to be accessed as Java classes and Java classes to be accessed as COM objects, thereby providing some support for using Microsoft software with Java.

EXERCISE 6-9

RMI vs. CORBA

Question What are the primary differences between RMI and CORBA, and for what is Internet Inter-ORB Protocol (IIOP) used?

Answer RMI and CORBA are both distributed-object technologies that support the creation, maintenance, and accessibility of objects. CORBA supports a language-independent approach to developing and deploying distributed objects. RMI is a Java-specific approach. IIOP is used to support communication between object request brokers such as CORBA via TCP/IP. RMI uses a stub that is a proxy for a remote object that runs on the client computer. RMI and CORBA use a skeleton as a proxy for a remote object that runs on the server. Stubs forward a client's RMIs (and their associated arguments) to skeletons, which forward them on to the appropriate server objects. Skeletons return the results of server method invocations to clients via stubs. The difference between RMI and CORBA is that the CORBA stubs access the ORB, and then the CORBA skeleton.

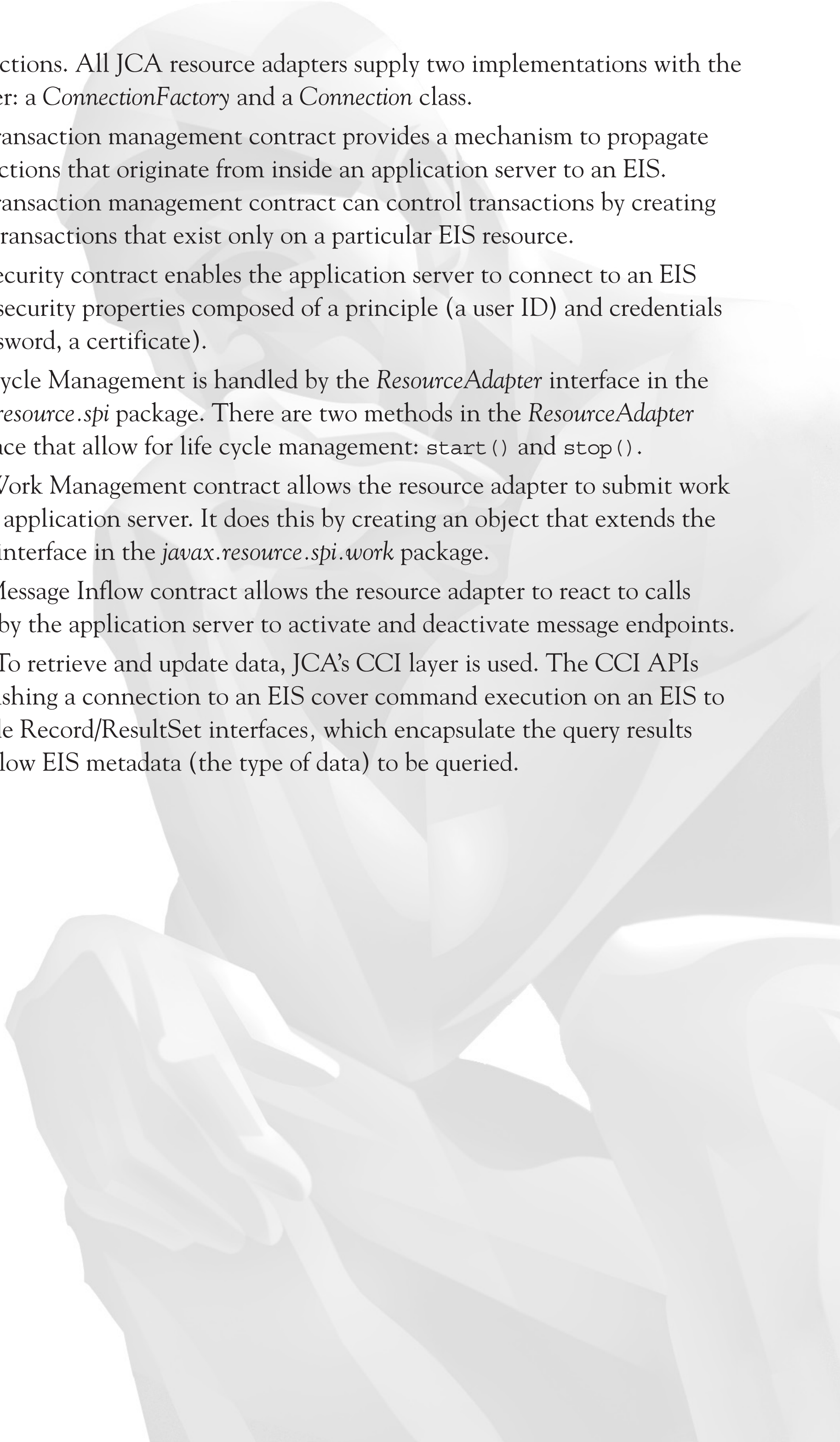
CERTIFICATION SUMMARY

The JCA is a specification for the Java platform that addresses the need to provide a standard architecture for integrating EIS. It complements the use of JNI and RMI to create a Java interface to a process running in its native domain.

✓ TWO-MINUTE DRILL

Distinguish Appropriate from Inappropriate Techniques for Providing Access to a Legacy System from Java Technology Code Given an Outline Description of That Legacy System

- ❑ The EAI facilitates the integration of EISs, or legacy systems, as they are also known. The classic means of communicating with an existing EIS has been a specialized adapter, which implements the support for communication with the EIS and provides access to EIS data and functions. Communication between an adapter and the EIS typically uses a protocol specific to the EIS.
- ❑ Another, more complex, form of an EIS adapter might do its "adaptation" work across diverse component models, distributed computing platforms, and architectures. For example, an EIS may develop a distributed adapter that includes the capability to perform remote communication with the EIS using Java RMI or CORBA.
- ❑ The JCA puts EAI into mainstream use by establishing a standard.
- ❑ The JCA comprises a resource adapter, connection management contracts, transaction management contract, security contract, and the CCI.
- ❑ A JCA resource adapter is specific to an EIS (Tibco) and is contained in a RAR file. The RAR is composed of the JAR files and native libraries required to deploy the resource adapter on a JEE container.
- ❑ A JCA adapter interacts with a JEE server via system contracts. Seven types of system contracts can be used:
    - ❑ Connection management
    - ❑ Transaction management
    - ❑ Security
    - ❑ Life Cycle Management (JCA 1.5)
    - ❑ Work Management Contracts (JCA 1.5)
    - ❑ Message Inflow (JCA 1.5)
    - ❑ Transaction Inflow Contracts (JCA 1.5)
- ❑ The connection management contract describes the interaction between a JEE container and the adapter with respect to pooling and tearing down



connections. All JCA resource adapters supply two implementations with the adapter: a *ConnectionFactory* and a *Connection* class.

❑ The transaction management contract provides a mechanism to propagate transactions that originate from inside an application server to an EIS. The transaction management contract can control transactions by creating local transactions that exist only on a particular EIS resource.

❑ The security contract enables the application server to connect to an EIS using security properties composed of a principle (a user ID) and credentials (a password, a certificate).

❑ Life Cycle Management is handled by the *ResourceAdapter* interface in the *javax.resource.spi* package. There are two methods in the *ResourceAdapter* interface that allow for life cycle management: `start()` and `stop()`.

❑ The Work Management contract allows the resource adapter to submit work to the application server. It does this by creating an object that extends the *Work* interface in the *javax.resource.spi.work* package.

❑ The Message Inflow contract allows the resource adapter to react to calls made by the application server to activate and deactivate message endpoints.

❑ CCI: To retrieve and update data, JCA's CCI layer is used. The CCI APIs establishing a connection to an EIS cover command execution on an EIS to provide Record/ResultSet interfaces, which encapsulate the query results and allow EIS metadata (the type of data) to be queried.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all the choices carefully because there might be more than one correct answer. Choose all correct answers for each question.

Distinguish Appropriate from Inappropriate Techniques for Providing Access to a Legacy System from Java Technology Code Given an Outline Description of That Legacy System

1. For a system consisting of exclusively Java objects, which distributed technology would be most appropriate for communication?

A. CORBA

B. RMI

C. JNDI

D. JavaBeans

2. Which of the following are true about the Interface Definition Language (IDL)?

A. Interfaces between CORBA objects can be specified using IDL.

B. Applications can be implemented using IDL.

C. Interfaces described in IDL can be mapped to other programming languages.

D. Stubs and skeletons are written in IDL.

3. An object that implements the interfaces *java.rmi.Remote* and *java.io.Serializable* is being sent as a method parameter from one JVM to another. How would it be sent by RMI?

A. RMI will serialize the object and send it.

B. RMI will send the stub of the object.

C. Both A and B throw an exception.

4. The RMI compiler rmic runs on which of the following files to produce the stub and skeleton classes?

A. On the remote interface class file

B. On the remote service implementation class file

C. On the remote service implementation Java file

D. On the remote interface Java file

5. Which distributed object technology is most appropriate for systems that consist of objects written in different languages and that execute on different operating system platforms?

A. RMI

B. CORBA

C. DCOM

D. DCE

6. Which of the following are used by Java RMI?

A. Stubs

B. Skeletons

C. ORBs

D. IIOP

7. Which of the following is not a tier of a three-tier architecture?

A. Client interface

B. Business logic

C. Security

D. Data storage

8. Which of the following is *not* true about RMI ?

A. RMI uses the Proxy design pattern.

B. RMI uses object serialization to send objects between JVMs.

C. The RMI Registry is used to generate stubs and skeletons.

D. The RMI client can communicate with the server without knowing the server's physical location.

SELF TEST ANSWERS

Distinguish Appropriate from Inappropriate Techniques for Providing Access to a Legacy System from Java Technology Code Given an Outline Description of That Legacy System

1. ☑ **B** is correct. RMI would be appropriate for communication between Java objects because it is built into the core Java environment. It is a built-in facility for Java, which allows you to interact with objects that are actually running on JVMs on remote machines on the network.
   ☒ **A, C,** and **D** are incorrect. CORBA is more extensive than RMI. Unlike RMI, objects that are exported using CORBA can be accessed by clients implemented in any language with an IDL binding. RMI is much more simple and straightforward than CORBA because it supports only Java objects. So where the facilities of CORBA are not required, it is preferable to go for RMI. JNDI and JavaBeans are not distributed object technologies.

2. ☑ **A** and **C** are correct. Interfaces between CORBA objects can be specified using IDL, but it is a language that can be used only for interface definitions. It cannot be used to implement applications.
   ☒ **B** and **D** are incorrect. We use other languages to implement the interfaces written in IDL. Interfaces written in IDL can be mapped to any programming language. CORBA applications and components are thus independent of the language used to implement them. Stubs and skeletons are not written; they are generated by the IDL compiler. Stubs and skeletons would be in the same language as the corresponding client or server.

3. ☑ **B** is correct. When you declare that an object implements the *java.rmi.Remote* interface, RMI will prevent it from being serialized and sent between JVMs as a parameter. Instead of sending the implementation class for a *java.rmi.Remote* interface, RMI substitutes the stub class. Because this substitution occurs in the RMI internal code, one cannot intercept this operation.
   ☒ **A** and **C** are incorrect. If the object had not implemented *Remote*, it would have been serialized and sent over the network.

4. ☑ **B** is correct. The RMI compiler, rmic, can be used to generate the stub and skeleton files. The compiler runs on the remote service implementation class file.
   ☒ **A**, **C,** and **D** are incorrect.

5. ☑ **B** is correct. CORBA is the most appropriate object technology for systems that use objects written in different languages, and it supports a variety of operating system platforms.
   ☒ **A, C,** and **D;** each works with specific platforms.

6. ☑ **A** and **B** are correct. RMI uses stubs and skeletons.
   ☒ **C** and **D** are incorrect because ORBs and IIOP are used with CORBA.

7. ☑ **C** is correct. Security is not a tier of a three-tiered architecture.
   ☒ **A, B,** and **D** are tiers of a three-tiered architecture.

8. ☑ **C** is correct because it is *not* true about RMI. RMI uses the proxy design pattern in the stub and skeleton layer. In the proxy pattern, an object in one context is represented by another (the proxy) in a separate context. The proxy knows how to forward method calls between the participating objects. In RMI's use of the proxy pattern, the stub class plays the role of the proxy. RMI uses a technology called *Object Serialization* to transform an object into a linear format that can then be sent over the network wire. The RMI compiler, rmic, is used to generate the stub and skeleton files.
   ☒ **A, B,** and **D** are incorrect.