

CHAPTER

8

Interoperability Solutions from Third-Party Vendors

IN THIS CHAPTER:

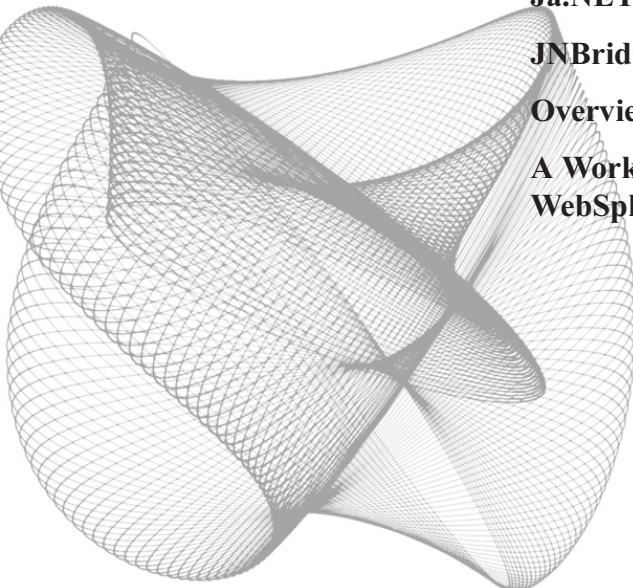
**Writing and Deploying Applications for Any
Platform**

Ja.NET and J-Integra

JNBridgePro: Infrastructure and Features

Overview of Installation

**A Working Example: JNBridgePro and
WebSphere 5.0**



In today's rapidly developing digital economy, businesses must reevaluate their methods of doing business in a distributed environment, preserving their heavy investment in legacy code while merging with the new technologies offered by Sun's Java platform and Microsoft's .NET Framework. J2EE and .NET provide numerous technologies to achieve interoperability, both internally and externally. For example, J2EE offers support for the constantly improving new technologies such as connectors and web services, whereas .NET provides multilanguage interoperability. Considering the success of the two diverse platforms and their unique approach to building distributed applications, how is it possible to take advantage of both technologies and build cross-platform applications?

Writing and Deploying Applications for Any Platform

Developing distributed applications in either J2EE or .NET presents a problem for the programmer: selecting a platform before writing the application. Which approach will solve a particular need? Java's philosophy is to write an application and deploy it on any platform containing a JVM. Only one caveat exists: the application *must* be written in the Java language.

Conversely, .NET offers developers multilanguage interoperability. A .NET application can run on any platform that has the .NET Framework on it. This is similar to a JVM, where Java bytecode can run on any platform where a copy of the JVM exists. However, currently only the Windows platforms have the Framework on them.

Enter third-party vendors. JNBridge LLC and Intrinsic Software are two major vendors of cross-platform technologies for J2EE and .NET. Both have recognized the predicament developers face when attempting to make their choice between J2EE and .NET technologies, and they offer interoperability solutions that share much in common:

- ▶ Support for enterprise application servers, including WebSphere, BEA WebLogic, Oracle9i, Borland Enterprise Server, and JBoss
- ▶ Support for HTTP and TCP/IP protocols
- ▶ Support for SOAP
- ▶ Support for binary messages

- ▶ Client-activated and server-activated objects
- ▶ Invocation of methods on Java objects from the CLR
- ▶ Invocation of methods on CLR objects from Java
- ▶ Support for passing Java/CLR objects by reference and by value as parameters/return values
- ▶ Marshaling objects by value or by reference
- ▶ Callbacks

It is no longer necessary for developers to make a critical choice. JNBridge LLC's product, JNBridgePro, and Intrinsic's Ja.NET bridge the gap between Java and .NET.

JNBridge's cross-platform technology is available in two versions: SE (Standard Edition) and EE (Enterprise Edition). The Enterprise Edition is covered in this chapter. But before going into the details of JNBridgePro, we'll take a brief look at Intrinsic's solutions.

Ja.NET and J-Integra

Ja.NET makes it possible to write clients for Enterprise JavaBeans in a .NET language targeting the .NET Framework. Using any language hosted by the Framework in conjunction with Ja.NET facilitates interoperability between Java objects or entity JavaBeans. Additional features permit reusing components written in Java within the .NET environment or vice versa. In essence, Ja.NET Java components act as though they are Microsoft .NET, and vice versa, because Ja.NET leverages .NET remoting.

Ja.NET provides a tool called GenJava to generate a Java proxy for .NET components. For example, access to an Internet Information Server (IIS) component from Java is easy. The Janetor tool configures the Ja.NET runtime. Then, Java clients can use the proxies to access a remote CLR component as though it were a local Java component. The tool can also generate a .NET component bearing proxies for Enterprise JavaBeans client-side classes. Then, Janetor generates a web application archive (WAR) file containing all web server-deployable files. Another beneficial feature permits the CLR client written in any language to access Enterprise JavaBeans as though they were local CLR components.

Intrinsic Software has another interop tool called J-Integra. It is a COM-Java tool employed for accessing ActiveX components as though they are Java objects. Conversely, accessing Java objects as though they are Microsoft .NET components is

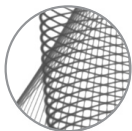
allowed. J-Integra works seamlessly with any Java Virtual Machine on any platform and requires no native code. Additionally, J-Integra speaks native DCOM and is layered over Remote Procedure Calls. J-Integra requires no JVM or additional software installs on the COM platform.

Ja.Net and J-Integra technologies are worth pursuing. You can freely download these solutions. Because URLs change frequently, I recommend that you search for them through your favorite search engine. Although this chapter does not examine Intrinsic Software's solutions, this does not reflect on the quality of their products. Check them out. They offer an excellent alternative to JNBridgePro.

JNBridgePro: Infrastructure and Features

JNBridgePro supports the creation of scalable and flexible distributed applications. Developers can access Enterprise JavaBeans, JMS, JNDI, and other services from .NET clients (Enterprise Edition only). JNBridgePro provides support for transactions via thread-true classes. The classes ensure they are managed by the same .NET-Java thread and guarantee data integrity. With JNBridgePro version 1.3.4 EE, developers can employ Java code in the .NET Framework as though Java is a .NET language. Fields, methods, and Java classes can be called from .NET classes without the caller being aware that the accessed classes are actually written in Java.

Microsoft .NET programmers can derive classes from Java. This feature allows the inherited classes to be written in a .NET language of choice, such as Visual Basic .NET, C#, J#, or managed C++ to name a few. The inheritance will be reflected on the Java side. Therefore, method calls overridden by derived .NET classes are redirected back to the .NET assembly. Subsequently, the Java classes can be linked to directly, or accessed over the network. This methodology supports a wide variety of platform architectures and communication mechanisms. An additional JNBridgePro benefit permits .NET classes to be implicitly accessed from Java code through the familiar Java callback listener method. (Java callbacks are discussed later in this chapter.)



NOTE

In order to work along with this chapter, an evaluation copy of JNBridgePro can be downloaded from the JNBridge web site. Because web site URLs can change quite frequently, simply type JNBridge into a search engine.

The JNBridgePro Infrastructure

Before examining the features that JNBridgePro offers developers, let's examine the infrastructure. The architectural details for the .NET-side are as follows:

- ▶ **.NET management** This is performed by the CLR.
- ▶ **Proxy generator** The JNB Proxy Generator creates the class proxies.
- ▶ **.NET classes** All .NET classes call the generated proxies. The proxy acts as the intermediary. In fact, this action is mandatory.
- ▶ **Generated proxies** The .NET assembly consists of generated proxy binary information.
- ▶ **Runtime component** This consists of a set of core proxies and classes to manage Java-.NET communications and references to Java objects. This can be a standalone application, an ASP.NET web application, or web service.

Details of the Java-side are as follows:

- ▶ **JVM** A standalone JVM or application server containing a J2EE servlet container
- ▶ **Java runtime component** SOAP or Fast Binary Protocol based on .NET remoting
- ▶ **Object reference table** Points to individual Java classes

JNBridgePro includes JNBProxy, a graphical user interface and proxy generator, and JNBCore, the interface that manages communications between .NET and the JVM. The JNBridgePro runtime architecture consists of a set of Java binaries (JAR or class files) containing the legacy Java code that runs on a JVM and Java Platform (J2SE) or (J2EE). This is the environment in which the Java code executes.

The JNBCore module manages communications between native-side proxies and the JVM. The JNBCore represents a uniform interface to either the .NET remoting framework that includes SOAP, the XML-based mechanism for invoking methods remotely on objects, or the faster binary protocol.

Residing on top of the JNBCore module is a set of native-side proxy classes that make up a .NET assembly, through which the .NET code accesses Java binaries. There must be *one proxy class for each Java class* accessed by .NET code. The

proxy's invocation interface matches the corresponding Java class. In order to create a corresponding Java object, the .NET code invokes the appropriate constructors on the proxy class and procures a reference to the new proxy object. To access a member field or method in a Java class, the .NET code invokes the corresponding field accessor or method in its associated proxy class.

Primitive values or references to proxy objects can be passed as parameters or returned as values. Exceptions thrown by Java objects are thrown back to the .NET code that invokes the proxies.

JNBProxy can be instructed to generate proxies for Java classes accessed by .NET code. This includes superclasses, exceptions, interfaces, return values, and parameters, or for that matter, any subset of them in between. Generating proxies for the maximum set of classes provides fine-grained granularity and maximum flexibility.

JNBridgePro Features

The key features offered by JNBridgePro include the following:

- ▶ Cross-platform and cross-language support and portability
- ▶ Class-level interoperability
- ▶ Large-scale distributed application support

Cross-Platform and Cross-Language Portability

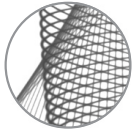
The ability to integrate Microsoft .NET with Java classes enables Java classes to function as though they are in reality a .NET-supported language. This technology requires no change to the Java code base. In fact, JNBridge LLC suggests developers can even make use of third-party libraries to achieve class integration. Despite Java and .NET class intermingling, the Java code remains cross-platform portable and conforms to existing J2EE standards.

Java code can run on a .NET machine or, for that matter, any machine supporting the JVM or a J2EE-compliant application server. An additional benefit is the ability to call Java classes and methods from any .NET-targeted Framework language.

Class-Level Interoperability

Class-level interoperability provides access to Java objects and classes from .NET via proxies generated by the proxy generation tool JNBProxy. This tool lets developers explore the interfaces and functionality of the available Java classes, allowing them to determine through a visual interface which classes should be exposed to .NET. Once the decision is made, JNBProxy generates the required

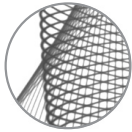
proxies and writes them to a .NET assembly DLL file. Therein lies the key to understanding how JNBridge Pro functions.

**NOTE**

Once the proxy is generated, .NET invokes all Java classes, methods, and fields on the proxy interface.

As you may recall from Chapter 5, the .NET assembly is made up of two essential components: the manifest and the assembly itself. The manifest contains the metadata about the assembly. The assembly contains version information and the name of the assembly as well as the application binaries.

The first portion of the assembly references all external assemblies required by the current assembly to execute the application successfully. The second segment of the assembly enumerates each module contained within the assembly. It is possible to load the assembly at runtime and procure a list of all types contained within a module. This includes both .NET and Java methods, events, fields, and properties. This is achieved by means of reflection.

**NOTE**

Each class whose functionality is exposed causes a proxy of the same name to be generated. Generated proxy members (constructors, methods, and fields) correspond to the members of the Java class underlying the proxy. Microsoft .NET interacts directly with the proxies rather than with the actual underlying Java classes.

Developers can call methods, access fields, create objects, access static and instance members of Java classes, and catch exceptions thrown by Java classes through the proxies.

Passing by Reference vs. Passing by Value

Beginning with version 1.3, JNBridgePro supports both reference and value objects. Passing by reference is faster than passing by value. A JNBridgePro reference is always smaller than the corresponding object. Moreover, repeated access to members of a reference object requires frequent trips between .NET and Java. In contrast, passing an object by value may take longer because the value object is larger. However, subsequent member accesses are much quicker because the data resides locally. Therefore, round-trips between Java and .NET are avoided. This enhances performance. It is always important to evaluate and determine which method offers increased performance. It always depends on application requirements.

Class-Level Interoperability Facilitates Using Java Callback Methods

Class-level interoperability allows developers to use Java's event listener methods. The .NET code can be called from Java via the standard Java callback mechanism, allowing the Java code to remain portable. For example, EJB containers are responsible for managing entity beans. Containers interact with entity beans by calling a bean's management or life cycle methods as necessary. These methods represent the bean's callback methods that only the container invokes, not the caller. These methods allow the container to alert a bean when middleware events occur, such as when an entity bean is going to be persisted to storage.

Java Data Types Are Automatically Converted to .NET Data Types

Java primitive types, strings, and arrays are automatically converted to corresponding .NET primitives, strings, and arrays. In contrast, the .NET primitives, strings, and arrays can be passed as parameters to Java classes and are subsequently converted to their Java counterparts.

With version 1.3, JNBridgePro automatically maps between native Java and .NET collection objects. Java ArrayLists, LinkedLists, Vectors, and HashSets are converted to .NET ArrayLists when they are returned from a .NET call to a specified Java method. Conversely, ArrayLists are converted from .NET to Java when they are passed as parameters in a call from .NET to a Java method. JNBridgePro maps between Java Hashtables, HashMaps, and .Net Hashtables as well.

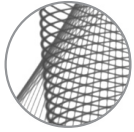
Inheritance, Implementation, and Class Proxies

.NET classes can inherit from Java classes by inheriting from the Java classes' proxies. Calls to any methods that the new class does not override are directed to the underlying Java class. Additionally, .NET classes may implement Java interfaces by implementing their proxies.

Java-.NET Class Integration

Once a proxy DLL is generated, the developer adds it to the current project. This allows the developer to access Java classes from .NET, a technique of considerable significance. Remember, the proxy DLL is written to the .NET assembly, the place where all information about Java classes, methods, and fields reside.

How is it possible to construct a Java object from .NET code? Simply call a constructor that corresponds to the Java constructor bearing the same name signature of the associated proxy class. In order to invoke an instance method on a Java object, the .NET code invokes the appropriate method or accesses the specified field *on the corresponding proxy object*.

**NOTE**

All .NET interaction with Java occurs through the generated proxy for a specified Java class. Every Java class contains a corresponding proxy.

Invoking a static method or field on a Java class calls the corresponding proxy class. Finally, if a Java method throws an exception, the .NET method catches the exception. Additionally, if the proxy method is no longer referenced, it is garbage collected, providing there are no further references to the object.

Overview of Installation

JNBridgePro includes a comprehensive set of evaluation and installation guides. This includes a PDF readme file as well as documentation for installations of the following containers:

- ▶ JNBridgePro with Borland Enterprise Server 5.0
- ▶ JNBridgePro with JBoss - Tomcat
- ▶ JNBridgePro with JRun
- ▶ JNBridgePro with Oracle9iAS
- ▶ JNBridgePro with Sybase EAserver 4.x
- ▶ JNBridgePro with WebLogic 6.1
- ▶ JNBridgePro with WebLogic 7.0
- ▶ JNBridgePro with WebSphere 4.0
- ▶ JNBridgePro with WebSphere 5.0

A little later in the chapter, we will look at an example using WebSphere's 5.0 server edition.

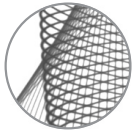
The general procedure for installing JNBridgePro is as follows:

1. Run the installer `jnbSetup1_1ee.msi` or `jnbSetup1_1se.msi` on the development machines where proxies are generated.
2. Configure the communications on the appropriate .NET and Java machine(s).
3. Copy the appropriate component and configuration files to the respective .NET-side and Java-side machines, and modify the configuration files as directed for runtime proxy-use deployment.

For an understanding of the specifics involved in installation, let's review the related topics as they are presented in the JNBridge guide.

Architectural Elements

When JNBridgePro is used to interoperate .NET and Java code, the .NET code runs on a .NET Common Language Runtime (CLR) and Java code runs on a Java Virtual Machine (JVM). Every application using JNBridgePro consists of one or more instances of the CLR and one or more JVM instances.



NOTE

Here, the set of all .NET CLR's in an application are collectively referred to as the ".NET-side," while the set of all instances of the JVM are referred to as the "Java-side."

The .NET-side and the Java-side can run on the same machine or on different machines connected by a network. The .NET-side must reside on a Microsoft .NET supported machine; the Java-side can reside on any machine with a standards-compliant JVM.

Configuring the .NET-Side

A copy of the .NET Framework must reside on the .NET-side. In addition, the .NET classes using JNBridgePro require the .NET-side to have a copy of `jnbshare.dll`, as well as a copy of the DLL file containing the generated proxies. These two DLL files must reside in the same folder as the application using them.

The configuration file `jnbproxy.config` must reside in the same folder as `jnbshare.dll`. If HTTP/SOAP is used for communications between .NET and Java, `jnbproxy.config` must contain a copy of `jnbproxy_http.config`. Conversely, if the binary protocol is used, `jnbproxy.config` must contain a copy of `jnbproxy_tcp.config` supplied with the JNBridgePro software distribution. Additionally, the configuration file must be edited to indicate the name or IP address of the host on which the Java classes reside, as well as the port number on which the Java-side listens for messages. If both the .NET-side and the Java-side reside on the same machine, the host name "localhost" may be used.

Since the .NET Framework supports distributed computing, the .NET-side can reside on several different machines, where classes communicate with each other through .NET remoting. If this scenario exists, each .NET-side machine communicating with the Java-side must contain a local copy of the generated proxy DLL file, `jnbshare.dll`, and `jnbproxy.config`.

Configuring the Java-Side

A copy of the Java Runtime Environment (JRE) must reside on the Java-side. If the Java-side resides on more than one machine, each respective machine must contain a JRE copy. The Java-side must also include any Java classes being accessed and must therefore have a copy of `jnbcore.jar` on each machine where the Java-side resides. The locations of the Java classes and `jnbcore.jar` must be in either the `CLASSPATH` environment variable or must be supplied to the Java runtime when invoked. Moreover, the Java-side must contain a copy of `jnbcore.properties`. Depending on whether communication is via HTTP/SOAP or binary protocol, `jnbcore.properties` should have a copy of one of the prototype properties files, `jnbcore_http.properties` or `jnbcore_tcp.properties`. Edit the Ports field in the properties file to indicate the port on which the Java-side should listen to messages from the .NET-side.

The Java-side for an application can reside on more than one machine, with the restriction that .NET classes sitting on one machine may only communicate with the Java classes on a single machine.

About Communications Protocols

JNBridgePro supports communication between .NET and Java code using two communications protocols:

- ▶ Proprietary binary protocol
- ▶ HTTP/SOAP communications protocol

The binary protocol is fast and efficient but may not penetrate corporate firewalls; the HTTP/SOAP protocol is slower, but will pass through firewalls. JNBridgePro recommends using the faster proprietary binary protocol.

If communications between .NET and Java employ HTTP/SOAP, a copy of a JAXP-compliant XML package must reside on the Java-side. Any such package may be used; one such package is located on Sun's JAXP site (<http://java.sun.com/xml/jaxp/index.html>). Installation details vary for each package. Programmers should consult instructions included with the selected package. To use HTTP/SOAP communications, the locations of the files in the JAXP package must either appear in the `CLASSPATH` environment variable, or they must be supplied to the Java runtime when invoked.

Table 8-1 lists JNBridgePro components and related configuration files.

Component	File Name	Description
JNBProxy GUI version	jnbproxygui.exe	Generates .NET proxies enabling .NET classes to communicate with Java classes.
JNBProxy command-line version	jnbproxy.exe	Represents the command-line version of the proxy generation tool.
JNPBCommon	jnbpcommon.dll	Contains functionality shared by the GUI and command-line versions of JNBProxy.
Generated proxies	*.dll	A .NET assembly contains the generated proxies built especially for the application.
JNBShare	jnbshare.dll	A .NET assembly contains core functionality used by generated .NET proxies.
JNBCore	jnbcore.jar	A Java JAR file that manages communications for both Java and .NET classes, as well as the object's life cycle—that is, creation, use, and object destruction of Java objects created by calls from .NET objects.
Registration tool	registrationTool.exe	Registers and licenses JNBShare on additional machines where the proxy generation tool has not been installed.
.NET-side proxy configuration file	jbnproxy.config	Specifies on the .NET-side, the protocol, host, and port used to communicate between .NET classes and Java classes.
Java-side configuration file	jnbcore.properties	Specifies on the Java side, the protocol and port number to communicate between the user's .NET classes and the Java classes.

Table 8-1 *JNBridgePro Development Components*

Executing the Installer

To install the JNBridgePro development components for proxy generation, double-click on the installer file `jnbSetup1_1ee.msi` or `jnbSetup1_1se.msi`, and follow the instructions, selecting one of the following options:

- ▶ Development configuration (proxy generation tools, plus Java and .NET runtime components)
- ▶ Deployment configuration (Java and .NET runtime components only)

In either case, the installer installs the files `jnbshare.dll`, `jnbcore.jar`, and `registrationTool.exe`, as well as sample configuration files and additional documentation. If the programmer selects the development configuration, the files `jnbproxy.exe`, `jnbproxygui.exe`, and `jnbpccommon.dll` are installed as well.

Configuring the Communications Protocol

As you already know, JNBridgePro supports communications between .NET classes running in a CLR and Java classes running in a JVM. The CLR and the JVM can reside on the same machine or on different machines. The JNBProxy proxy generation tool is simply a .NET application communicating with Java classes like any other .NET application utilizing JNBridgePro. Therefore, JNBProxy also requires a CLR and a JVM, existing on either the same or different machines.

1. Navigate to the folder in which `jnbproxy.exe` and `jnbproxygui.exe` were installed (for example, `C:\Program Files\JNBridge\JNBridgePro`).
2. Generate a new configuration file, `jnbproxy.config`, specifying the protocol (HTTP/SOAP or binary) used to communicate between .NET and Java classes.
 - ▶ For HTTP/SOAP, copy `jnbproxy_http.config` to `jnbproxy.config`.
 - ▶ For binary protocol, copy `jnbproxy_tcp.config` to `jnbproxy.config`.
3. Modify the new `jnbproxy.config` file in order to reflect the proper host name and port number for the Java runtime component.
 - ▶ Open `jnbproxy.config` using a text editor and locate the URL, which will either be `http://localhost:8085/JNBDispatcher`, for HTTP/SOAP, or `jtcp://localhost:8085/JNBDispatcher`, for the binary protocol.

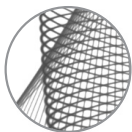
- ▶ Edit the host name (currently “localhost”) and port number (currently “8085”) to indicate the host on which the Java classes and the JNBCore component will reside and the port on which JNBCore will be listening for messages. If JNBCore resides on the same machine as the .NET classes, leave the host as “localhost.”

The default JNBCore is residing on the same machine as the .NET classes; JNBCore listens on port 8085. If this is acceptable, do not change the configuration file. Save it, and close the text editor.

4. Copy the jnbcore directory into a location of choice on the machine where the Java environment resides. This directory contains the Java runtime component jnbcore.jar and two properties files, jnbcore_http.properties and jnbcore_tcp.properties. This directory must be on the same machine as the Java classes accessed from .NET code. Add the JAR’s full path to the CLASSPATH environment variable.
5. Generate a new properties file, jnbcore.properties, corresponding to the protocol used to communicate between .NET classes and Java classes.
 - ▶ For HTTP/SOAP, copy jnbcore_http.properties to jnbcore.properties.
 - ▶ For binary protocol, copy jnbcore_tcp.properties to jnbcore.properties.
6. On the Java machine, open jnbcore.properties using a text editor. Edit the value of the port to agree with the value chosen in jnbproxy.config. Save the file, and close the text editor. The resultant file should look like the following:

```
serverType=tcp
workers=5
timeout=10000
port=8085
```

If using HTTP/SOAP for communications between the .NET classes and Java classes, and a JAXP-compliant XML package is not installed, install one. If using the binary protocol, skip this step.



NOTE

Any JAXP-compliant XML package can be installed. Follow the installation instructions for the selected package. Following installation, add the JAR file’s full path to the CLASSPATH environment variable.

Improving Network Performance

By default, JNBridgePro sends out network packets between the .NET and Java sides as soon as they are created. In most cases, this leads to improved performance. However, this behavior means that typical JNBridgePro-generated network packets are small, and in some cases this could lead to network congestion and degraded performance. If network performance degradation occurs, turn off the NoDelay option so packets are aggregated before they are sent out. This may improve network performance. Typically, if calls or returns contain a large amount of data, turning NoDelay off may improve performance. The NoDelay option can also be controlled independently in the .NET-to-Java and Java-to-.NET directions. To turn the NoDelay option off in the .NET-to-Java direction, add the following to the .NET application's configuration file. For example, if the application is x.exe, create or open the file x.exe.config in the same folder as x.exe and add the following to the file:

```
<configuration>
<appSettings>
<add key="JNBridge_TCP_NoDelay" value="false"/>
</appSettings>
</configuration>
```

To turn the NoDelay option off in the Java-to-.NET direction, add the following line to the jnbcore.properties file: **nodelay=false**.

Starting Java for Proxy Generation

JNBProxy uses the Java reflection API to discover information about Java classes for which it is generating proxies. To do this, a JVM must be running that contains the JNBCore component and the Java classes for generating proxies.

JNBProxy users (both the GUI and command-line versions) may choose to have the Java-side started automatically or manually. If the Java-side resides on some other machine, JNBProxy will be unable to start; therefore, a manual start is required. To manually start the Java-side, issue the following command from the command-line:

```
java -cp classpath com.jnbridge.jnbcore.JNBMain /props propFilePath
```

where classpath must include the following:

- ▶ The Java classes for which proxies are generated (and their supporting classes).
- ▶ jnbcore.jar.
- ▶ The JAR files for the XML package (if using HTTP/SOAP communication).
- ▶ The `-cp` classpath option can be omitted, in which case the required information must be present in the CLASSPATH environment variable: `propFilePath` is the full file path of the file `jnbcore.properties`.

The folder containing the `java.exe` executable must be in the system's search path (typically described in the PATH environment variable). If not, specify the full path of `java.exe` on the command line.

When starting the Java-side manually, you will see the following output if using the binary protocol:

```
JNBCore v1.2
Copyright 2002, JNBridge
creating binary server
```

Or the following output will be seen if HTTP/SOAP is used:

```
JNBCore v1.2
Copyright 2002, JNBridge
creating http server
```

Configuring the System for Proxy Use

To deploy JNBridgePro runtime components for proxy use (that is, on machines other than ones containing the proxy generator), copy and run the installer on those machines, and select Deployment Configuration when requested by the installer. Also create versions of the configuration files `jnbproxy.config` and `jnbcore.properties` as described in the earlier section “Configuring the Communications Protocol.” JNBridgePro searches for the configuration file `jnbproxy.config` in the following folders:

- ▶ It looks first in the folder where `jnbshare.dll` is located.
- ▶ If not located there, it looks in the folder `<System-drive>:\inetpub\wwwroot`, where `<System-drive>` is the drive on which the running system is installed (typically C, but not always). If the proxies are called from ASP.NET, place `jnbproxy.config` in this folder.

If `jnbshare.dll` is not found there, JNBridgePro looks in the folder `<System-drive>:\` (the root folder of the drive on which the system is installed, typically `C`, but not always). Any user accessing proxies generated by JNBridgePro must specify Read and Write access to the folder `<System-drive>:\Documents and Settings\AllUsers\Application Data\Microsoft\Crypto\RSA\MachineKeys`.

If the user does not have access to this folder, the proxies cannot be used. This is only an issue on systems running the NTFS file system. On systems running other file systems, users should automatically have this access.

Configuring Proxies for Use with ASP.NET

If the developer is calling the generated proxies from ASP.NET, place the configuration file `jnbproxy.config` in the folder `<System-drive>:\inetpub\wwwroot`, where `<Systemdrive>` is the drive on which the running system is installed (typically `C`). If the ASP.NET machine is running on an NTFS file system, make sure the user account has Read and Write access to the folder `<System-drive>:\Documents and Settings\All Users\ApplicationData\Microsoft\Crypto\RSA\MachineKeys`.

If the required access permission is not assigned, the ASP.NET program will not be able to use the proxies.

Starting a Standalone JVM for Proxy Use

JNBridgePro applications accessing Java classes and objects from .NET must have an installed and running Java-side. This can be either a standalone JVM, or a J2EE application server. You will find information on configuring a J2EE application server to work with JNBridgePro in the *Users' Guide*. A standalone JVM must contain the JNBCore component and Java classes before generating the proxies.

To start the Java-side manually in a standalone JVM, specify the following command-line command:

```
java -cp classpath com.jnbridge.jnbcore.JNBMain /props propFilePath
```

where `classpath` must include the following:

- ▶ Java classes and their supporting classes for generating proxies
- ▶ `jnbcore.jar`
- ▶ JAR files for the XML package (if using HTTP/SOAP communication)

If the `-cp classpath` option is omitted, include the following information required in the `CLASSPATH` environment variable: `propFilePath`, the full file path for `jnbcore.properties`. The folder containing the `java.exe` executable must be in the system's search path (typically described in the `PATH` environment variable). If not, specify the full path of `java.exe` on the command line.

When starting the Java-side manually, you will see the following when using the binary protocol:

```
JNBCore v1.2
Copyright 2002, JNBridge
creating binary server
```

or the following if using HTTP/SOAP:

```
JNBCore v1.2
Copyright 2002, JNBridge
creating http server
```

Running the Java-Side Under Nondefault Security Managers

The Java-side will work under the default security manager (that is, the one used if no security manager is explicitly set). If a security manager is set, permissions may need to be granted explicitly to allow `jnbcore.jar` to run. In the appropriate copy of the file `java.policy`, add the following lines:

```
grant codebase "file://location_of_jnbcore.jar_goes_here"
{
  permission java.lang.RuntimePermission "*", "accessDeclaredMembers";
  permission java.net.SocketPermission "*", "accept,resolve";
}
```

`jnbcore` will now run with all required permissions. If a security manager has been set, you may also need to grant permissions for your Java classes running on the Java-side.

A Working Example: JNBridgePro and WebSphere 5.0

Let's observe an example provided by JNBridgePro called `BasicCalculatorEJB`. It demonstrates how to access Enterprise JavaBeans from C#. The EJBs will be running on IBM WebSphere Application Server 5.0. Assume that JNBridgePro 1.3

or later and a .NET framework are installed on a Windows machine. Also assume that WebSphere 5.0, along with the accompanying examples, exists on a compatible machine, which may be either the same machine as the .NET machine or a different machine reachable on the network.

Creating jnbcore.war

Create the jnbcore.war file as follows:

1. Create a blank folder to hold the WAR file components. Call this folder <warroot>.
2. Create folders <war-root>/WEB-INF and <war-root>/WEB-INF/lib.
3. In <war-root>/WEB-INF/lib, place a copy of jnbcore.jar (copied from the JNBridgePro installation folder) and a copy of the BasicCalculatorEJB.jar file found in the Demos/J2EE-Examples/WebSpere5.0 folder in the installation.
4. In <war-root>/WEB-INF, place copies of the files jnbcore.properties and web.xml found in the Demos/J2EE-Examples/WebSphere5.0 folder in the installation. Open a command-line window and navigate to <war-root>.
5. Run the command **jar cf jnbcore.war WEB-INF**.
6. Install jnbcore.war by starting the WebSphere Application Server and opening the Administration console. Install jnbcore.war in the usual way, using the supplied default values. There are two places where you must supply data:
 - ▶ When prompted for the context root, enter **/jnbcore**.
 - ▶ When asked for the JNDI name for the EJB-reference, enter **WSsamples/BasicCalculator**.
7. Make sure the box asking whether the EJBs should be deployed remains unchecked. The BasicCalculator EJB has already been deployed.
8. Once jnbcore.war is deployed, save it to the master configuration and start it.
9. Exit the Administration console. Shut down WebSphere Application Server for the moment so that its jnbcore does not interfere with the jnbcore used by JNBProxy.

Building the Proxy DLL

Build the proxy assembly DLL as follows:

1. Start a copy of the JNBProxy GUI-based tool.
2. Select the menu item Project.Java options. Verify that the box Start Java Automatically is checked, and the Java configuration data is correct.

3. Close the dialog box.
4. Select the menu item Project>Edit Classpath and add the following files to the classpath:
 - ▶ <JNBridgePro-WebSphere5-demo-folder>/BasicCalculatorEJB.jar (it contains the classes BasicCalculator and BasicCalculatorHome, used in the C# code).
 - ▶ j2ee.jar in <was-server-root>/lib (it contains the classes EJBHome, EJBObject, and other EJB-related and JNDI-related support classes, including classes Context, InitialContext, and CreateException, used in the C# code). Close the dialog box. Create a proxy for RemoteException. The Java class is found in the local J2SDK file; no additional JAR file is needed.
5. Select the menu item Project>Add Classes from Classpath and add the following classes. Make sure the box Include Supporting Classes is checked for each class added.
 - ▶ com.ibm.websphere.samples.technologysamples.ejb.stateless.basiccalculatorejb.BasicCalculator
 - ▶ com.ibm.websphere.samples.technologysamples.ejb.stateless.basiccalculatorejb.BasicCalculatorHome
 - ▶ javax.naming.Context
 - ▶ javax.naming.InitialContext
 - ▶ javax.naming.NamingException
 - ▶ javax.rmi.PortableRemoteObject
 - ▶ java.rmi.RemoteException
 - ▶ javax.ejb.CreateException
6. Click OK to add the classes and all supporting classes to the Environment pane. This will take a few minutes. Check all the items in the environment (use the menu item Edit.Check All in Environment), and click on the Add button to move them all to the Exposed Proxies pane. Select the menu item Project.Build to build the proxy assembly. Choose an assembly name and click OK. The assembly will be written to a DLL file.

Building and Running the Client Application

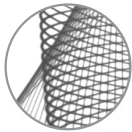
These tasks can be similarly accomplished using the .NET Framework SDK:

1. In Visual Studio .NET, create a new C# console application.

2. Replace the generated file `Class1.cs` with the file `Class1.cs` located in the `Demos/J2EE-Examples/WebSphere5.0` folder. You may wish to examine this file to see how the Java classes and EJBs are accessed from .NET.
3. Add references to the previously generated proxy assembly DLL and to the assembly `jnbshare.dll` found in the JNBridgePro installation folder.
4. Build the project.
5. Find the file `jnbproxy_tcp.config` in the JNBridgePro installation folder and make a copy of it in the current project build directory. Rename the file `jnbproxy.config`.
6. Restart WebSphere Application Server and verify that `jnbcore.war` has been deployed by looking in the standard output log or by opening the Administration console. Then run the .NET application. Output showing the results of calculations will be displayed.

The BasicCalculatorEJB Sample Files

Let's look at some of the files for this example. First, the C# file. Here is the code:



NOTE

JNBridge LLC has given permission to quote the following code examples, which you can use as templates for developing your applications.

```
using System;
using java.lang;
using java.rmi;
using javax.ejb;
using javax.naming;
using javax.rmi;
using com.ibm.websphere.samples.technologysamples._
ejb.stateless.basiccalculatorejb;
/*
In C#, using imports the class libraries
*/

// JNBridgePro example C# code for WebSphere 5.0.
// Copyright 2003, JNBridge, LLC

namespace ws5Demo
{
    /// <summary>
    /// Summary description for Class1.
```

```

/// </summary>
class Class1
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
        BasicCalculatorHome basicCalculatorHome = null;

        // get the context
        // get the calculator home interface
        try
        {
            System.Console.WriteLine("JNDI stuff...");
            Context ctx = new InitialContext();
            System.Console.WriteLine("Got context");
            java.lang.Object homeObject =
ctx.lookup("WSsamples/BasicCalculator");
            System.Console.WriteLine("Got home object");

            basicCalculatorHome =
                (BasicCalculatorHome) homeObject;
System.Console.WriteLine("Got calculator home interface");
        }
        catch( NamingException e )
        {
            System.Console.WriteLine("NamingException: " + e);
            return;
        }
        catch( java.lang.Exception e2 )
        {
            System.Console.WriteLine("Generic exception: " + e2);
            return;
        }

        // create the remote interface
        BasicCalculator basicCalculator = null;
        try
        {
            basicCalculator = basicCalculatorHome.create();
            Console.WriteLine("got remote interface");
        }
        catch ( javax.ejb.CreateException e1)
        {
            Console.WriteLine("CreateException: " + e1);

```

```

        return;
    }
    catch( java.lang.Exception e2 )
    {
        Console.WriteLine("Generic exception: " + e2);
        return;
    }

    // do some calculations
    try
    {
        double arg1 = 2.5;
        double arg2 = 3.9;
        double result;
// In C# we box the parameter into an object using {}, one for each argument
        result = basicCalculator.makeSum(arg1, arg2)
        Console.WriteLine("{0} + {1} = {2}", arg1, arg2, result);
        result = basicCalculator.makeDifference(arg1, arg2);
        Console.WriteLine("{0} - {1} = {2}", arg1, arg2, result);
        result = basicCalculator.makeProduct(arg1, arg2);
        Console.WriteLine("{0} * {1} = {2}", arg1, arg2, result);
        result = basicCalculator.makeQuotient(arg1, arg2);
        Console.WriteLine("{0} / {1} = {2}", arg1, arg2, result);
    }
    catch(java.lang.Exception ex)
    {
        Console.WriteLine("Generic exception
        in calculation: " + ex);
        return;
    }
}
}
}
}

```

Next, here is the Web-app file:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//
DTD Web Application 2.2//EN" "http://java.sun.com
/j2ee/dtds/web-app_2_2.dtd">
<web-app>
    <display-name>JNBridgePro Servlet</display-name>

```

```

<servlet>
  <servlet-name>JNBServlet</servlet-name>
  <servlet-class>com.jnbridge.jnbcore.JNBServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<ejb-ref>
  <ejb-ref-name>WSsamples/BasicCalculator</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>com.ibm.websphere.samples.technologysamples.
ejb.session.basiccalculatorejb.BasicCalculatorHome</home>
  <remote>com.ibm.websphere.samples.technologysamples.
ejb.session.basiccalculatorejb.BasicCalculator</remote>
  <ejb-link>BasicCalculator</ejb-link>
</ejb-ref>
</web-app>

```

JNBridgePro has included a Demo package with source code for you to use as a template for your own cross-platform applications. Here are the three classes: Book.Java, Author.Java, and DemoDBConnection.Java, in that order:

```

//Book.Java
package Demo2;
// Copyright 2002, JNBridge LLC
import java.sql.*;
import java.util.*;

public class Book
{
    public String title;
    public String publisher;
    public String year;

    public Book(String title, String publisher, String year)
    {
        this.title = title;
        this.publisher = publisher;
        this.year = year;
    }

    public static Book[] getBooks()
    {
        try
        {
            Connection conn = DemoDBConnection.getConnection();

```

```

        Statement stmt = conn.createStatement();
        ResultSet titles = stmt.executeQuery
("SELECT Title, Publisher, Year FROM Titles
    ORDER BY Title");

        Vector v = new Vector();
        while (titles.next())
        {
            v.add(new Book(titles.getString("Title"),
titles.getString("Publisher"), titles.getString("Year")));
        }

        Book[] results = new Book[v.size()];
        for (int i = 0; i < v.size(); i++)
        {
            results[i] = (Book) v.get(i);
        }

        return results;        // for now
    }
    catch(Exception e)
    {
        e.printStackTrace();
        System.out.println(e);
        return null;
    }
}

public static Book[] getBooks(String firstName, String lastName)
{
    try
    {
        Connection conn = DemoDBConnection.getConnection();
        Statement stmt = conn.createStatement();
        String query = "SELECT Title,
Publisher, Year FROM Titles, Authors " +

"WHERE FirstName = \' " + firstName + "\' AND " +

"LastName = \' " + lastName + "\' AND Authors.ID =
Titles.AuthorID ORDER BY Title";

```

```

        ResultSet titles = stmt.executeQuery(query);

        Vector v = new Vector();
        while (titles.next())
        {
            v.add(new Book(titles.getString("Title"),
titles.getString("Publisher"),
titles.getString("Year")));
        }

        Book[] results = new Book[v.size()];
        for (int i = 0; i < v.size(); i++)
        {
            results[i] = (Book) v.get(i);
        }

        return results;        // for now
    }
    catch(Exception e)
    {
        e.printStackTrace();
        System.out.println(e);
        return null;
    }
}
}

```

Here is Author.Java:

```

//Author.Java
package Demo2;
// Copyright 2002, JNBridge LLC
import java.sql.*;
import java.util.*;

public class Author
{
    public String firstName;
    public String lastName;

    public Author(String firstName, String lastName)
    {
        this.firstName = firstName;
    }
}

```

```

        this.lastName = lastName;
    }

    public static Author[] getAuthors()
    {
        try
        {
            Connection conn = DemoDBConnection.getConnection();
            Statement stmt = conn.createStatement();
            ResultSet authorNames =
stmt.executeQuery("SELECT LastName, FirstName
FROM Authors ORDER BY LastName");

            Vector v = new Vector();
            while (authorNames.next())
            {
                v.add(new Author
(authorNames.getString("FirstName"),
authorNames.getString("LastName")));
            }

            Author[] results = new Author[v.size()];
            for (int i = 0; i < v.size(); i++)
            {
                results[i] = (Author) v.get(i);
            }

            return results;        // for now
        }
        catch(Exception e)
        {
            e.printStackTrace();
            System.out.println(e);
            return null;
        }
    }
}

```

DemoDBConnection.Java follows:

```

//DemoDBConnection.Java
package Demo2;
// Copyright 2002, JNBridge LLC

```

```

import java.sql.*;

public class DemoDBConnection
{
    private static DemoDBConnection theDBConnection
    = new DemoDBConnection();
    private Connection theConnection;

    public static Connection getConnection()
    {
        return theDBConnection.theConnection;
    }

    private DemoDBConnection()
    {
        try
        {
            // load the database driver
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

            // allocate a connection object
            theConnection = DriverManager.getConnection
            ("jdbc:odbc:BookDemo");
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}

```

Next, the three stubs are presented. First, the Book stub:

```

//public class Book
package Demo2;
public class Book {
    // Fields
    public String title;
    public String publisher;
    public String year;

    // Constructors

```

```

public Book(String string, String string1, String string2) { }

// Methods
public static Book[] getBooks() { return null;}
public static Book[] getBooks
(String string, String string1) { return null;}
}

```

Next, the following code is the Author stub:

```

package Demo2;
public class Author {
    // Fields
    public String firstName;
    public String lastName;

    // Constructors
    public Author(String string, String string1) { }

    // Methods
    public static Author[] getAuthors() { return null;}
}

```

The following code represents the DemoDBConnection stub:

```

//This stub is DemoDBConnection
package Demo2;

// Imports
import java.sql.Connection;

public class DemoDBConnection {

    // Fields
    private static DemoDBConnection theDBConnection;
    private Connection theConnection;

    // Constructors
    private DemoDBConnection() { }

    // Methods
    public static Connection getConnection() { return null;}
}

```

Next, we present the WebForm1.cs file:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using Demo2;

// Copyright 2002, JNBridge LLC

namespace Demo3
{
    /// <summary>
    /// Summary description for WebForm1.
    /// </summary>
    public class WebForm1 : System.Web.UI.Page
    {
        protected System.Web.UI.WebControls.Label Label1;
        protected System.Web.UI.WebControls.ListBox ListBox1;
        protected System.Web.UI.WebControls.Label Label2;
        protected System.Web.UI.WebControls.ListBox ListBox2;

        private void Page_Load(object sender, System.EventArgs e)
        {
            // Put user code to initialize the page here
        }

        #region Web Form Designer generated code
        override protected void OnInit(EventArgs e)
        {
            //
            // CODEGEN: This call is required
            //
            by the ASP.NET Web Form Designer.
            //
            InitializeComponent();
            base.OnInit(e);
        }

        /// <summary>
        /// Required method for Designer support - do not modify

```

```

    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
        this.ListBox1.Load +=
newSystem.EventHandler(this.ListBox1_Load);
this.ListBox1.SelectedIndexChanged += new
System.EventHandler(this.ListBox1_SelectedIndexChanged);
        this.Load += new System.EventHandler(this.Page_Load);
    }
#endregion

private void ListBox1_Load(object sender, System.EventArgs e)
{
    if (ListBox1.Items.Count > 0)
    {
        return;        // don't add anything more
    }

    // load the authors
    Author[] authors = Author.getAuthors();
    for (int i = 0; i < authors.Length; i++)
    {
        ListItem li = new ListItem();
        li.Text = authors[i].lastName + ", " +
authors[i].firstName;
        ListBox1.Items.Add(li);
    }
}

private void ListBox1_SelectedIndexChanged
(object sender, System.EventArgs e)
{
    // clear listBox2
    ListBox2.Items.Clear();

    // if nothing selected, return
    ListItem li = ListBox1.SelectedItem;

    if (li == null)
    {
        return;
    }

    // last, first names
    string name = li.Text;

```

```

int separator = name.IndexOf(",");
string lastName = name.Substring(0, separator);
string firstName = name.Substring(separator+2);

// look up books
Book[] titles = Book.getBooks(firstName, lastName);

// write out books
for (int i = 0; i < titles.Length; i++)
{
    ListItem li2 = new ListItem();
    string theBook = titles[i].title + "
        (" + titles[i].publisher + ", "
        + titles[i].year + ")";
    li2.Text = theBook;
    ListBox2.Items.Add(li2);
}
}
}

```

A copy of the Access database including the Author and Book may be found in the Demo2 folder included with the JNBridgePro installation package.

In summary, JNBridgePro and Intrinsic offer viable cross-platform business solutions for every business scenario. Finally, developers have a tool to facilitate true interoperability between J2EE version 1.3.1 and Microsoft .NET.